universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Evaluierung von Large Language Models zur Python-Codegenerierung für Geodatenverarbeitungsaufgaben

verfasst von | submitted by

Giacomo Joggerst BSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2025

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 856
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Kartographie und Geoinformation
Betreut von | Supervisor:	Ass.-Prof. Mag. Dr. Andreas Riedl

Danksagung

Ich möchte mich bei Ass.-Prof. Mag. Dr. Andreas Riedl für die engagierte Betreuung meiner Masterarbeit bedanken sowie bei meinen Studienkollegen, die mir dabei halfen, die Motivation für die Fertigstellung der Arbeit hochzuhalten. Außerdem möchte ich mich bei der Firma AGGM bedanken, bei der ich einige Fähigkeiten, auf denen meine Arbeit basiert, erlernen und Kenntnisse aus dem Studium vertiefen durfte. Besonders möchte ich mich auch bei meiner Mutter bedanken, ohne die mir das Studium nicht möglich gewesen wäre und bei meiner Freundin Anne, die mich immer wieder im Prozess der Arbeit mental unterstützt hat.

Inhaltsverzeichnis

Abbildungen

Tabellen

Abkürzungen

AI	Artificial Intelligence
API	Application Programming Interface
BLEU	Bilingual Evaluation Understudy
bzw.	beziehungsweise
d.h.	das heißt
DL	Deep Learning
EPSG	European Petroleum Survey Group
GIS	Geoinformationssystem
K.I	Künstliche Intelligenz
LLM	Large Language Model
ML	Machine Learning
NLP	Natural Language Processing
OSM	Open Street Map
ROUGE	Recall-Oriented Understudy for Gisting Evaluation
z.B.	zum Beispiel

Kurzfassung

Diese Masterarbeit untersucht die Fähigkeiten großer Sprachmodelle (LLMs) zur Lösung komplexer Geodatenverarbeitungsaufgaben innerhalb von Geoinformationssystemen (GIS). Angesichts der rasanten Entwicklung von LLMs ist es entscheidend, ihr Potenzial sowie ihre Grenzen in spezialisierten Bereichen wie der Geodatenanalyse zu verstehen. Die Studie umfasst drei Testszenarien, die reale GIS-Aufgaben simulieren, darunter die Verarbeitung räumlicher Daten, die symbolbasierte Visualisierung von Attributen und die Integration von Koordinatensystemen. Jedes Szenario bewertet das räumliche Denken, die Datenumwandlung und das kontextuelle Verständnis der Modelle anhand von Metriken wie Accuracy (Genauigkeit) und Pass@k. Durch den Vergleich der Leistung mehrerer LLM‘s in diesen Aufgaben soll die Forschung Erkenntnisse über ihre Effektivität und Zuverlässigkeit für Geodaten-Anwendungen liefern. Die Ergebnisse zeigen, dass LLMs ein erhebliches Potenzial zur Automatisierung bestimmter GIS-Funktionen besitzen, jedoch bei komplexeren räumlichen Aufgaben Herausforderungen bestehen. Diese Arbeit trägt zur aktuellen Diskussion über die Anpassung von LLMs an fachspezifische Anwendungen bei und gibt Empfehlungen für ihren zukünftigen Einsatz im GIS-Bereich.

Abstract

This thesis investigates the capabilities of large language models (LLMs) in solving complex geospatial data tasks within Geographic Information Systems (GIS). With the rapid development of LLMs, understanding their potential application and limitations in specialized fields like geospatial analysis has become crucial. The study explores three test scenarios that simulate real-world GIS tasks, including spatial data processing, attribute-based symbolization, and the integration of coordinate systems. Each scenario assesses the models' spatial reasoning, data transformation, and contextual understanding, using metrics such as accuracy and pass@k evaluations. By comparing the performance of multiple LLMs across these tasks, the research aims to provide insights into their effectiveness and reliability for geospatial applications. The findings suggest that while LLMs exhibit significant potential for automating certain GIS functions, challenges remain in complex spatial reasoning tasks. This study contributes to the ongoing discourse on adapting LLMs for domain-specific applications, offering recommendations for their future deployment in GI.

1 Einleitung

1.1 Motivation und Zielsetzung

Spätestens mit dem Release von Open AI's „ChatGPT" im November 2022 erlangten Large Language Models einen globalen Bekanntheitsgrad und ein neuer Meilenstein im Forschungsbereich der künstlichen Intelligenz wurde erreicht. Denn diese Computermodelle sind in der Lage, komplexe sprachliche Zusammenhänge zu verstehen und sogar funktionalen Programmiercode zu generieren. Während frühere Sprachmodelle oft nur auf spezialisierte Aufgaben oder Domänen zugeschnitten waren, zeigten diese neuartigen Modelle beeindruckende Generalisierungsfähigkeiten. Dadurch eröffnen sich eine Vielzahl an Anwendungsmöglichkeiten in verschiedenen Forschungs- und Anwendungsfeldern. Auch im Bereich der Geoinformatik bergen diese Modelle ein enormes Potential, da sich hier durch Programmierung hohe Effizienzgewinne erreichen lassen. Denn in diesem Fachgebiet, das sich unter anderem mit der Modellierung, Analyse und Visualisierung von räumlichen Daten befasst, ist die Automatisierung von entscheidender Bedeutung, da bei der Arbeit häufig ein großer Umfang an wiederkehrenden und teilweise komplexen Verarbeitungsschritten anfällt. Beispielsweise müssen Daten aus verschiedenen Quellen zusammengeführt werden, Koordinatensysteme harmonisiert, komplexe räumliche Analysen durchgeführt oder Geometrien bereinigt werden. Traditionell erfordern Geoprozessierungs-Skripts spezifische Programmierkenntnisse und die Beherrschung von spezialisierten Softwarebibliotheken wie PyQGIS oder ArcPy. Dies stellt insbesondere für Anwender ohne tiefgehende Programmierfähigkeiten eine große Hürde dar. LLM's könnten hierbei eine transformative Rolle spielen, indem sie die Erstellung von Geoprozessierungs-Skripts vereinfachen. Beispielsweise können Anwender durch einfache Textaufforderungen in natürlicher Sprache ein Skript erhalten, was sich direkt in einer GIS-Umgebung ausführen lässt, anstatt mit hohem Zeitaufwand Funktionen und Klassen in Dokumentationen nachschlagen und komplexe Syntaxregeln berücksichtigen zu müssen. Aus diesem Grund wird sich diese Arbeit auf eine Evaluierung von Large Language Models hinsichtlich ihrer Fähigkeit, Programmcode für Geoprozessierungen zu generieren, konzentrieren. Es soll zudem aufgezeigt werden, welche Fehlerquellen und Einschränkungen bestehen. Es stellt sich die Frage, wie robust und zuverlässig LLM's in einem hochspezialisierten technischen Umfeld wie der Geoinformatik agieren. Durch diese Evaluierung soll ein Beitrag zur Diskussion über die Rolle von LLM`s in der Geoinformatik geleistet werden, mit dem Ziel, eine Brücke zwischen K.I und praktischer GIS-Anwendung zu schlagen.

1.2 Problemstellung und Stand der Forschung

Die rasante Entwicklung und zunehmende Verbreitung von Large Language Models (LLMs) wie Open AI's Chat GPT, Googles Gemini und Metas LLaMa haben diese Modelle zu einem festen Bestandteil der modernen computergestützten Textverarbeitung gemacht. Ihre Fähigkeit, kontextbezogene, sprachliche Aufgaben zu lösen und auch Programmier-Code zu generieren wurde umfassend demonstriert und erforscht (Brown et al., 2020) & (Chen et al., 2021) . Die GIScience-Gemeinschaft integrierte in den letzten Jahren K.I in die Forschung und Anwendungen, was zu GeoAI führte, einem Teilbereich, der sich auf die Schnittstelle von K.I und GIScience fokussiert (Janowicz, Gao, McKenzie, Hu & Bhaduri, 2020). Dennoch ergeben sich Herausforderungen für die Anwendung in einzelnen wissenschaftlichen Disziplinen wie der Geoinformationsverarbeitung, die nicht nur mit Text, sondern multimodalen Daten umgehen muss und domänenspezifisches Wissen erfordert (Mai et al., 2023). In der aktuellen Forschung finden sich bisher theoretische und konzeptionelle Erörterungen: Viele der aktuellen Arbeiten diskutieren die theoretischen Grundlagen und konzeptionellen Rahmen von Geospatial Foundation Models und Large Language Models. Sie konzentrieren sich auf die Entwicklung und Verbesserung von Modellen, die breit angelegte georäumliche Aufgaben abdecken können, einschließlich der Erstellung von georäumlichen Wissensgraphen und der Integration von räumlichen Daten in bestehende Modelle (Xie et al., 2023). Weiters wird die allgemeine Anwendung von LLMs in Geodatenanalysen erforscht, wie LLMs in der Geodatenanalyse verwendet werden können, z.B. für die Kartierung oder die Analyse georäumlicher Daten. Diese Studien tendieren dazu, die Anwendbarkeit und Effektivität der Modelle in breiten, oft weniger spezifischen Szenarien zu bewerten (Tao & Xu, 2023). Auch wird untersucht, wie Large Language Models mit geografischen Daten umgehen. Sie analysieren die Verarbeitung von Ortsnamen, Koordinaten und räumlichen Zusammenhängen. Die Forschung konzentriert sich auf die Genauigkeit dieser Systeme bei geografischen Informationen (Manvi et al., 2023) (Mooney, Cui, Guan & Juhász, 2023). Zudem richtet sich aktuelle Forschung auf zukunftsorientierte Ansätze: Einige der Arbeiten sind visionärer Natur und beschäftigen sich mit der langfristigen Entwicklung von GeoAI-Modellen, die eine umfassende Palette von Aufgaben in der Geoinformatik abdecken könnten und weitgehend autonom, also mit wenig menschlicher Intervention funktionieren (Mai et al., 2023) & (Li & Ning, 2023). Außerdem gibt es aktuelle Forschung, in der Evaluierungs-Benchmarks entwickelt wurden, um die Leistungsfähigkeit von Large Language Models für Codegenerierung zu testen (Chen et al., 2021) auch speziell bezogen auf Geodatenverarbeitungsaufgaben (Gramacki, Martins & Szymański, 2024). Die Verarbeitung von Geodaten erfordert spezialisierte Kenntnisse und ist oft zeitaufwändig. In Bezug auf die Funktionsweise von Large Language Models werden verschiedene Ansätze erforscht, die sich auf die Verbesserung der Leistung der Modelle bei komplexeren Problemlösungsaufgaben konzentrieren (Yao et al., 2023).

Da viele Anwender Schwierigkeiten haben, komplexe Python-Skripte zu schreiben, die für die Geodatenverarbeitung notwendig sind, ist ein zentraler Aspekt meiner Masterarbeit die Untersuchung, ob LLMs wie GPT-4 in der Lage sind, diese Aufgaben zu automatisieren und die Genauigkeit und Effizienz der Geodatenverarbeitung zu verbessern. Während LLMs in der Lage sind, Code zu generieren, bleibt die Frage, ob dieser Code korrekt und funktionsfähig ist, insbesondere für spezialisierte Aufgaben wie die Geodatenverarbeitung. Es ist darum entscheidend zu überprüfen, ob die generierten Python-Skripte für die Geodatenverarbeitung präzise genug sind, um in realen Anwendungen eingesetzt zu werden. Unterschiedliche LLMs haben verschiedene Architekturen, Spezifikationen und Trainingsdaten. Es ist unklar, welches Modell für diese spezifische Aufgabe am besten geeignet ist. Ein Ziel meiner Arbeit ist es, die Leistung verschiedener LLMs bei der Generierung von Python-Code zur Geodatenverarbeitung zu vergleichen und festzustellen, ob es signifikante Unterschiede gibt.

Die folgende Tabelle 1 gibt einen Überblick über aktuelle relevante Forschung in diesem Kontext.

No.	Autor	Titel	Untersuchung	Resultate
1	MAI et al. (2023)	Towards a Foundation Model for Geospatial Artificial Intelligence.	Konzeptioneller Rahmen Herausforderungen und Potentiale	Paper zeigt das Potenzial von Foundation Models für GeoAI auf und stellt ein Framework für deren Umsetzung vor, betont aber auch die Herausforderungen und Risiken die damit einhergehen.
2	TAO (2023)	Mapping with ChatGPT	Anwendbarkeit der Modelle in breiten, weniger spezifischen Szenarien	ChatGPT hat Potenzial, Kartenerstellungsprozesse zu vereinfachen und zu verbessern, jedoch in der aktuellen Entwicklungsstufe noch Grenzen, insbesondere bei der direkten Kartenvisualisierung
3	GRAMACKI et al. (2024)	Evaluation of Code LLMs on Geospatial Code Generation	Entwicklung eines Evaluierung Benchmark-Sets zur Bewertung von LLM Code für “geospatial tasks”	Benchmark-Dataset mit spezifischen geospatial Programmieraufgaben. Test von mehreren aktuellen Code LLM’s, einfache geodatenverarbeitungsaufgaben wurden besser gelöst als Multi-Step-Aufgaben mit spezialisiertem Tool-Wissen Erfordernis
4	CHEN et al. (2021)	Evaluating Large Language Models Trained on Code	Benchmark und Evaluierungsmetriken für LLM Code	Verwendet HumanEval Methode für LLM Benchmark

5	LI et al. (2023)	Autonomous GIS: the next-generation AI-powered GIS	GIS Agent für automatisierte GIS Prozesse	Entwicklung eines Prototyps, der autonome GIS-Operationen mithilfe eines LLMS ausführt.
6	MANVI et al. (2024)	GeoLLM "Extracting Geospatial Knowledge from Large Language Models"	Methode um geospatial knowledge aus LLMs zu extrahieren mittels auxiliary map data. Finetunes GPT 3.5.	Aktuelle LLMS nutzen grundlegendes geographisches Wissen, ihre Genauigkeit und Konsistenz variiert aber je nach Modell und Methode
7	YAO et al. (2023)	Tree of Thoughts: Deliberate Problem Solving with LLM's	Untersucht, wie LLM's in komplexeren Problemlösungsaufgaben verbessert werden können.	Stellt den Tree of Thoughts (ToT) Ansatz vor, der das Erkunden mehrerer Lösungswege ermöglicht. Dieser Ansatz fördert bewusstes Entscheiden durch das Erforschen und Vergleichen mehrerer Denkrichtungen innerhalb eines „ToT" (Entscheidungsbaum)
8	MOONEY et al. (2023)	Towards Understanding the Geospatial Skills of ChatGPT	Untersuchung, wie gut ChatGPT fähig ist, geografische Informationssysteme zu verstehen, indem es einem GIS-Examen unterzogen wird.	GPT-4 und 3.5 konnten grundlegende Fragen gut beantworten, zeigten aber Schwächen bei komplexeren Aufgaben, besonders bei Berechungen oder Fragen die implizite räumlichen Überlegungen erforderten
9	Vaswani et al. (2017)	Attention is all you need	Vorstellung der Transformer Architektur, die einen Paradigmenwechsel in der Sequenzmodellierung darstellt	Der Transformer übertraf bestehende Modelle bei zwei maschinellen Übersetzungsaufgaben

Tabelle 1: Überblick über aktuell relevante Forschung im Bereich GeoAI (Eigene Darstellung 2024)

1.3 Relevanz der Arbeit

Diese Masterarbeit beschäftigt sich mit der Evaluierung der Fähigkeiten von Large Language Models (LLMs) zur Python-Codegenerierung für spezifische Anwendungen in der Geodatenverarbeitung. Dieses Forschungsthema ist relevant, weil es die Schnittstelle zwischen fortgeschrittenen Technologien in der künstlichen Intelligenz und den praktischen Anforderungen der Geoinformatik adressiert (Mai et al. 2023). Geodaten spielen eine zunehmend kritische Rolle in einer Vielzahl von Anwendungsbereichen, von der Umweltüberwachung und der Stadtplanung bis hin zur Katastrophenhilfe und Ressourcenmanagement. (Goodchild et al. 2023) Die Fähigkeit, effizient und präzise Code zu generieren, der diese Daten verarbeiten kann, ist daher von großer praktischer Bedeutung und könnte Arbeitsprozesse massiv beschleunigen. Durch das Verstehen, welche Modelle effektiv Code generieren können, der direkt in Geoinformation-Anwendungen verwendet wird, lassen sich Arbeitsabläufe optimieren. Weiters vereinfacht die Nutzung von Natural Language die Analysen und Prozesse und steht somit einem breiteren Anwenderpublikum zur Verfügung und kann zu erheblichen Zeit- und Kosteneinsparungen führen (Tao et al. 2023). Diese Arbeit soll die Brücke zwischen theoretischer Forschung und praktischer Anwendung schlagen und aufzeigen, inwiefern LLMs einen wertvollen Beitrag leisten können und welche Weiterentwicklungen nötig sind, um ihre Effektivität und Anwendbarkeit in der Geoinformationsverarbeitung weiter zu verbessern.

1.4 Zielsetzung und Forschungshypothese

Das Hauptziel dieser Masterarbeit ist es, die Fähigkeiten von Large Language Models (LLMs) zur Generierung von Python-Code für die Verarbeitung von Geodaten systematisch für die Durchführung von definierten spezifischen Workflows zu evaluieren. Dies umfasst mehrere Unterziele, die sowohl die technische Leistungsfähigkeit der Modelle als auch deren praktische Anwendbarkeit in geospezifischen Aufgabenstellungen betreffen: Bei der technischen Evaluierung erfolgt eine Untersuchung der Genauigkeit und Funktionalität des von verschiedenen LLMs generierten Python-Codes. Dies beinhaltet die Bewertung, wie gut die Modelle in der Lage sind, aus einem Prompt, formuliert in Natural Language Python Code zu generieren, der einen spezifischen Workflow korrekt ausführt. In der vergleichenden Analyse wird ein direkter Vergleich zwischen mehreren relevanten LLMs durchgeführt, um festzustellen, welches Modell am effektivsten für geospezifische Programmieraufgaben geeignet ist. Letztlich soll eine Identifikation der Limitierungen der Vorgehensweise durchgeführt werden und Verbesserungspotential erschlossen werden. Hier werden die spezifischen Anforderungen und Herausforderungen identifiziert, die mit der Nutzung von LLMs für die Codegenerierung für Geodatenverarbeitung verbunden sind. Die Überlegungen in

den vorherigen Abschnitten zeigen klar das Potenzial von LLMs. Diese können angewendet werden um Geodatenverarbeitungsaufgaben effektiv lösen. Daher wird für die vorliegende Arbeit folgende Forschungshypothese aufgestellt:

„Die zu testenden Large Language Models unterscheiden sich signifikant hinsichtlich ihrer Fähigkeit, korrekten Python-Code für räumliche Analyseaufgaben zu generieren. Dabei zeigen sich Unterschiede in der Genauigkeit (Accuracy) und Zuverlässigkeit (Pass@k) der Modelle, insbesondere bei verschiedener Komplexität und Mehrstufigkeit der Aufgaben."

Um die definierten Ziele erreichen zu können, werden folgende Arbeitsfragen bearbeitet:

- Inwieweit können Large Language Models effektiv Python-Code zur Lösung klassischer Geodatenverarbeitungsaufgaben generieren und welche Faktoren beeinflussen ihre Leistung in diesem Kontext?"

- Welche Leistungsunterschiede zeigen sich bei der Verwendung von den ausgewählten LLM`s für die spezifischen Geodatenverarbeitungsaufgaben?

1.5 Methodische Vorgehensweise

Im ersten Schritt wird durch eine vergleichende Literaturrecherche Basiswissen und der aktuelle Stand der Forschung erarbeitet. Die Auswahl der LLMs basiert auf Benchmarks, Verfügbarkeit und Modellcharakteristik. Geodatenverarbeitungsaufgaben wie Feature-Manipulation und räumliche Analysen dienen als Testfälle. Zur Evaluierung wird der HumanEval-Benchmark in Form der Pass@k-Metrik eingesetzt, um syntaktische Korrektheit, Zuverlässigkeit und funktional/logische Anforderungen zu bewerten. Außerdem wird die Genauigkeit (Accuracy) der generierten Lösungen ermittelt. In den Experimenten wird Zero-Shot-Prompting zur Simulation realistischer Anwendungsfälle verwendet. Die Ergebnisse bieten eine quantitative Bewertung der Modellleistung und analysieren Limitierungen. Abschließend werden die Ergebnisse zusammengefasst sowie zukünftige Forschungsansätze diskutiert.

1.6 Struktur der Arbeit

Diese Arbeit ist in mehrere aufeinander aufbauenden Kapitel unterteilt, um eine systematische Untersuchung zu ermöglichen. Nach der Einleitung bildet das 2. Kapitel den theoretischen Hintergrund. Hier werden die notwendigen Grundlagen vermittelt, die für das Verständnis der Arbeit notwendig sind. Zunächst gibt es hier eine Einführung in die Grundlagen der Künstlichen Intelligenz mit dem Fokus auf Large Language Models und deren Anwendung im Bereich Natural Language Processing (NLP). Es folgt die Verknüpfung von Künstlicher Intelligenz mit der wissenschaftlichen Disziplin der Geoinformatik und es wird auf die Rolle von Programmierung in Geoinformationssystemen eingegangen und ihre Bedeutung im Zusammenhang mit Automatisierung erörtert. Kapitel 3 beschreibt die Methodik, nach der in der Arbeit vorgegangen wurde. Hier wird zunächst die Auswahl der Large Language Modelle, Deepseek Coder, Claude 3.5 Sonett und GPT o1 Preview begründet und die erarbeiteten Testszenarien beschrieben, einschließlich Datengrundlage und Testumgebung. Diese Szenarien bilden einen realen Anwendungsfall im Bereich der immer relevanten Energiebranche ab. Zusätzlich erfolgt noch zum besseren Verständnis der Testmodelle eine technische Beschreibung dieser. Folgend wird auf die Bewertungs- und Evaluierungsmetriken Pass@k und Accuracy, die zur Bewertung der Modelle herangezogen werden, eingegangen. Abschließend wird die Durchführung der einzelnen Tests angeführt. Im Folgenden, 4. Kapitel werden die Ergebnisse aus den Tests präsentiert und es findet sich eine Analyse der Fehler, die im Code der Large Language Modelle vorhanden waren. Im nachfolgenden Unterkapitel „Ergebnisse und Diskussion“ werden die Ergebnisse in einen größeren Kontext gesetzt und mit Resultaten von anderen Forschungsarbeiten verglichen. Außerdem wird beschrieben, an welche Grenzen und Herausforderungen die durchgeführte Vorgehensweise gestoßen ist. Im abschließenden Kapitel der Arbeit folgt noch eine kurze Zusammenfassung der zentralen Erkenntnisse der Arbeit und reflektiert deren Bedeutung für die Forschung und Praxis, sowie ein Ausblick auf zukünftige Forschung in diesem Kontext.

2 Theoretischer Hintergrund

2.1 Grundlagen der Künstlichen Intelligenz und Large Language Models

Die Künstliche Intelligenz kurz: „K.I“ befasst sich mit der Nachbildung von menschlichem Intelligenzverhalten unter der Verwendung von Computerprogrammen. Dieser Begriff wurde im Jahr 1956 von John McCarthy geprägt. K.I umfasst die Schaffung von Systemen, die dazu in der Lage sind, komplexe Aufgaben mehr oder weniger eigenständig zu lösen. (Künstliche Intelligenz - Lexikon der Neurowissenschaft, 2014) Die Hauptansätze der Künstlichen Intelligenz werden unterteilt in Machine Learning und Deep Learning. Machine Learning wird als ein Zweig der Künstlichen Intelligenz beschrieben, der auf statistischen Methoden oder numerischen Optimierungstechniken basiert, um Modelle aus Daten abzuleiten, ohne dass jeder Parameter oder Berechnungsschritt explizit programmiert werden muss. Es wird häufig für Klassifikation, Clustering und Vorhersage verwendet. Deep Learning ist wie in Abbildung 1 visualisiert ein weiterer Teilbereich der K.I. Dieser Bereich hingegen konzentriert sich auf die Verwendung sogenannter neuronaler Netzwerke für Machine Learning-Aufgaben, diese Netzwerke sind dem menschlichen Gehirn nachempfunden. Deep Neural Networks (DNN) sind eine Art der Künstlichen Intelligenz, die mehrere Schichten zwischen einer Eingabe- und einer Ausgabeschicht verwenden und oft zu hochwertigen Ergebnissen führen können, weshalb Deep Learning in den letzten Jahren viel Aufmerksamkeit erhalten hat (Hu et al., 2019, S. 4).

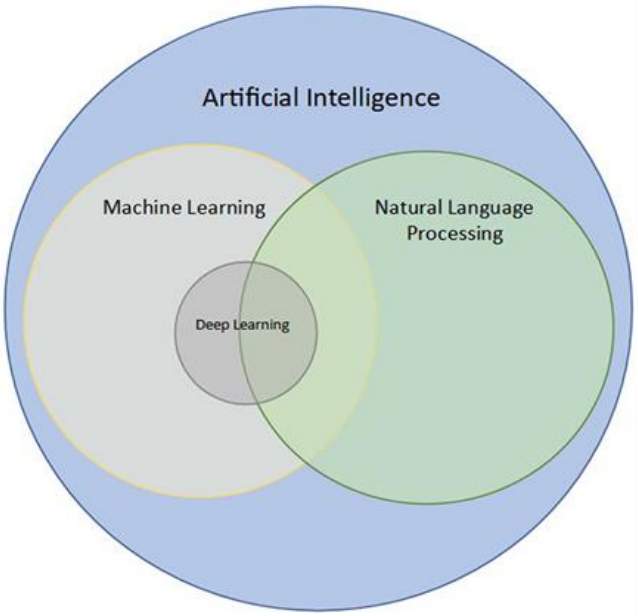

Abbildung 1: Venn-Diagramm visualisiert die Überschneidung der Felder: AI, ML, DL und NLP (PubMed, 2024, S. 2)

2.1.1 Natural Language Processing

Das Feld der Verarbeitung von natürlicher Sprache, wie es Large Language Models zu tun vermögen, nennt sich engl. „Natural Language Processing“ (NLP). LLMS‘s bauen auf den Konzepten und Algorithmen von NLP auf und sind daher für das Verständnis dieser von Bedeutung (Amaratunga, 2023, S. 18) Dieser Bereich konzentriert sich darauf, Computern zu ermöglichen, menschliche Sprache auf eine Weise zu verstehen, zu interpretieren und zu generieren (Amaratunga, 2023, S. 9). Natural Language Processing (NLP) ist ein Teilgebiet der künstlichen Intelligenz, wie in Abbildung 1 im Venn-Diagramm dargestellt. Es konzentriert sich darauf, Computern zu ermöglichen, menschliche Sprache auf eine Weise zu verstehen, zu interpretieren und zu generieren, die sowohl sinnvoll als auch nützlich ist. Das Hauptziel von NLP besteht darin, die Kluft zwischen menschlicher Sprache und dem Verständnis durch Computer zu überbrücken, sodass Maschinen natürliche Sprachdaten verarbeiten, analysieren und darauf reagieren können (Amaratunga, 2023, S. 9).

Die Geschichte des NLP reicht bis in die 1950er Jahre zurück. 1950 veröffentlichte Alan Turing einen Artikel mit dem Titel „Computing Machinery and Intelligence“, in dem er eine Methode vorschlug, um zu bestimmen, ob eine Maschine menschliche Intelligenz aufweist. Dieser vorgeschlagene Test, der heute als Turing-Test bekannt ist, gilt als Inspiration für frühe NLP-Forscher, die das Verständnis natürlicher Sprache durch Maschinen untersuchen wollten (Amaratunga, 2023, S. 9-10). Während der 1960er und 1970er Jahre stützte sich die NLP-Forschung vorwiegend auf regelbasierte Systeme. Ein frühes NLP-Programm war der ELIZA-Chatbot, der von Joseph Weizenbaum zwischen 1964 und 1966 entwickelt wurde. ELIZA nutzte Musterabgleich und einfache Regeln, um eine Unterhaltung zwischen dem Benutzer und einem Psychotherapeuten zu simulieren (Amaratunga, 2023, S. 10). In den 1970er und 1980er Jahren begann die NLP-Forschung, linguistische Theorien und Prinzipien zu integrieren, um die Sprache besser zu verstehen. Die Theorien von Noam Chomsky zur generativen Grammatik und zur Transformationsgrammatik beeinflussten frühe NLP-Arbeiten maßgeblich (Amaratunga, 2023, S. 11). Die linguistikbasierten Ansätze des NLP hatten jedoch ihre Grenzen. Die Komplexität der natürlichen Sprachen und die Vielzahl von Ausnahmen zu linguistischen Regeln machten es schwierig, umfassende und robuste NLP-Systeme allein auf Basis formaler Grammatiken zu entwickeln. Daher wurden diese Ansätze im Laufe der Zeit durch datengetriebene Methoden und statistische Modelle ergänzt und teilweise ersetzt (Amaratunga, 2023, S. 12).

2.1.2 Large Language Models

Das Kernstück der Untersuchungen dieser Arbeit bilden Large Language Models, kurz „LLM’s“, welche einen weiteren Teilbereich der Künstlichen Intelligenz bilden. Im folgenden Abschnitt werden die Grundlagen dieser Modelle erläutert und Konzepte erläutert, die für das Verständnis der Arbeit in den späteren Kapiteln notwendig sind. Folgend wird eine

Visualisierung (Abbildung 2) der verschiedenen Schritte eines LLM-Workflows aufgeführt. Dieser Workflow wird in den folgenden Unterkapiteln genauer erklärt.

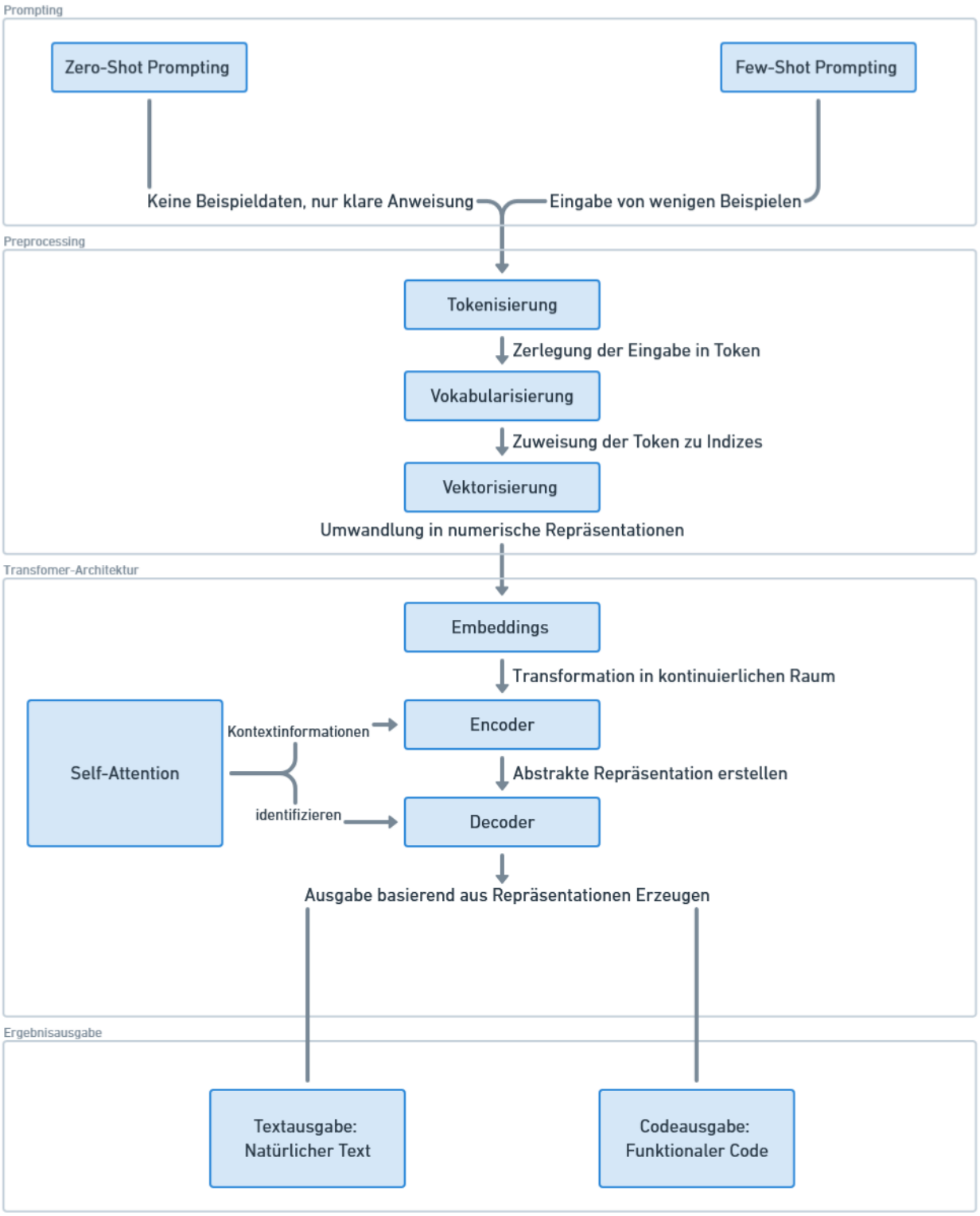

Abbildung 2: LLM-Workflow (Vaswani et al., 2017, S. 3) Eigene Überarbeitung 2025

2.1.2.1 Zero-Shot und Few-Shot Prompting

Zero Shot bezeichnet die Situation, wenn ein Sprachmodell eine Aufgabe, welche durch einen Prompt erhalten wurde, ohne jegliche Beispiele oder Demonstration der Aufgabe ausführt. Es soll also direkt und ohne spezifisches Training für diese Aufgabe eine Lösung generieren (Brown et al., 2020, S. 4). In Abbildung 3 wird ein Beispiel für diesen Vorgang visualisiert.

Abbildung 3: Visualisierung eines Zero-Shot Prompts (Brown, 2020, S. 7)

Few-Shot beschreibt ein Szenario, bei dem das Modell während des Prompts einige Beispiele für die jeweilige Aufgabe als Kontext erhält, ohne dass dabei allerdings die Gewichtung der Parameter angepasst werden und unterscheidet sich somit vom Fine Tuning (Brown et al., 2020, S. 6). In der Abbildung 4 findet sich ein Beispiel für dieses Vorgehen.

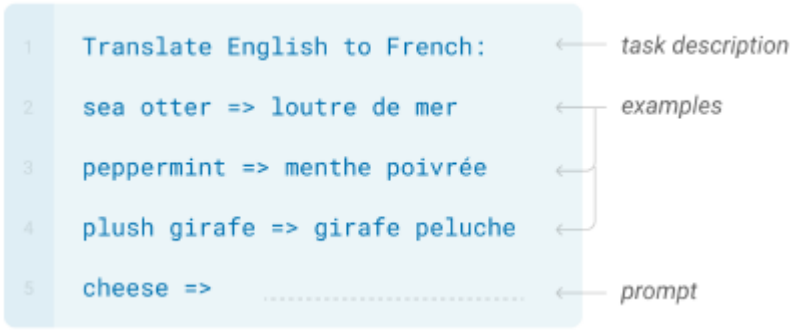

Abbildung 4: Visualisierung eines Few-Shot Prompts (Brown, 2020, S. 7)

2.1.2.2 Preprocessing: Tokenisierung

Hierbei handelt es sich um einen wichtigen Preprocessing Schritt, also Vorverarbeitung der Texte, die von LLMS verarbeitet werden. Mit Hilfe eines spezifischen Tokenizers der von Modell zu Modell variieren kann, wird Text oder eine Zeichenfolge in kleinere Einheiten, die sogenannten Tokens aufgeteilt (Abbildung 2). Nimmt man den Satz „Shapefile must die!“ als Beispiel, so könnte ein tokenisierter Satz folgendermaßen aussehen:

["Shapefile", "must", "die", "!"]

Innerhalb des Prozesses wurde also der Satz in einzelne Wörter aufgespalten und jegliche Interpunktion entfernt. Es resultiert eine Liste von Token, in der jedes Wort aus dem Satz als eigener Token vorkommt. Generell gibt es hier verschiedene Techniken von Tokenisierung, so kann dabei ein Satz in einzelne Wörter oder auch in noch kleinere Einheiten aufgespalten werden. Die Wahl des Tokenizers, also des Werkzeugs das diesen Arbeitsschritt ausführt, hängt dabei von der jeweiligen NLP-Aufgabe ab (Amaratunga, 2023, S. 20)

Die resultierenden Token können nun als sogenannte: „Embeddings“ weiterverarbeitet werden. Embeddings sind Vektorepräsentationen von Wörtern in einem Vektorraum, in diesem liegen ähnliche Wörter näher beieinander wodurch die semantische Bedeutung zwischen den Wörtern und die Beziehungen untereinander erfasst werden können

Es folgt ein Beispiel zur Bag-of-Words-Methode:

Nehmen wir Toblers berühmtes First Law of Geography: "Everything is related to everything else, but near things are more related than distant things."

Schritt 1: Tokenisierung

Die Tokens in diesem Satz sind:

["Everything", "is", "related", "to", "everything", "else", "but", "near", "things", "are", "more", "related", "than", "distant", "things"].

Schritt 2: Vokabularerstellung

Das Vokabular enthält alle einzigartigen Wörter aus dem tokenisierten Satz:

["Everything", "is", "related", "to", "else", "but", "near", "things", "are", "more", "than", "distant"].

Vokabulargröße: 12

Schritt 3: Vektorisierung

Nun wird der Satz als Vektor unter Verwendung des Vokabulars dargestellt. Der Vektor für den Satz sieht wie folgt aus:

[2, 1, 2, 1, 1, 1, 1, 2, 1, 1, 1, 1]

Der Vektor zeigt, dass die Wörter "everything", "related" und "things" zweimal im Satz vorkommen, während die anderen Wörter jeweils einmal vorkommen.

Wichtig ist zu bemerken, dass die Reihenfolge der Wörter in der Bag-of-Words-Darstellung verloren geht und der Satz ausschließlich anhand der Häufigkeit der darin enthaltenen Wörter dargestellt wird (Amaratunga, 2023, S. 26) Während die bisher vorgestellten Methoden in weniger komplexen Modellen Anwendung finden, wollen wir uns nun, man könnte sagen „Königsklasse“, der Sprachmodellierung widmen der sogenannten Transformer Architektur, wie sie auch in state of the art LLM’s wie GPT4 zur Anwendung kommt.

2.1.2.3 Transformer-Architektur

Die Transformer-Architektur war ein großer Fortschritt in der Forschung im Bereich NLP und wurde erstmals im Paper "Attention Is All You Need" von Vaswani et al. Im Jahr 2017 vorgestellt. Der große Vorteil gegenüber herkömmlichen Ansätzen wie beispielsweise „recurrent neural networks (RNN)“ liegt hierbei darin, dass sie nicht auf sequenzielle Verarbeitung limitiert ist sondern sogenannte „self attention“ also Selbstaufmerksamkeits-Mechanismen verwenden, wobei alle Positionen in einer Sequenz gleichzeitig betrachtet werden und damit auch Langzeitabhängigkeiten (Long Term Dependencies) eingeschlossen werden, das heißt auch die Abhängigkeiten und Beziehungen von eher weit entfernten Wörtern in einem Text werden

berücksichtigt und ermöglichen damit ein tieferes Verständnis von Kontext. Des Weiteren steigt bei herkömmlichen Ansätzen wie ConvS2S and ByteNet models die Anzahl der erforderlichen Operationen linear mit dem Abstand zwischen den Elementen wodurch sich diese Vorgehensweise ineffizienter gestaltet als bei der Transformer-Architektur, wo die Anzahl der notwendigen Operationen konstant bleibt (Amaratunga, 2023, S. 64). Generell besteht diese Architektur aus dem Encoder und dem Decoder. Die wichtigste Funktion des Encoders ist dann die Selbstaufmerksamkeit (Self Attention) für das einzelne Token gegenübereinander zu berechnen und „lernt" somit Muster und Beziehungen zwischen den Wörtern und gibt erweiterte Repräsentationen als Output heraus, welche dann im Decoder verwendet werden kann. Der Decoder ist dann für die Generierung eines Outputs zuständig und verwendet wiederum Selbstaufmerksamkeit, um zu verstehen, welche Teile der bisher generierten Sequenz wichtig sind, um das nächste Token vorherzusagen, siehe Abbildung 2 (Vaswani et al., 2017, S. 3–5)

2.2 Geoinformatik und Künstliche Intelligenz

Die wissenschaftliche Disziplin in deren Rahmen sich das Thema dieser Arbeit befindet ist zusätzlich zum Bereich der K.I. die „Geoinformatik". Sie bildet eine eigenständige Disziplin und ist nicht zu verwechseln mit dem Begriff: „GIS" der häufig synonym für diesen Bereich der Wissenschaft verwendet wird, sich aber im engeren Sinne auf die Nutzung von Geoinformationssystemen (GIS) als Werkzeug zur Verarbeitung von Geoinformationen bezieht (Lange, 2020, S. 1). Definieren lässt sich das Feld folgendermaßen, die Geoinformatik konzentriert sich darauf, informatikbasierte Methoden und Ansätze zu entwickeln und anzuwenden, um räumliche Probleme und Fragestellungen zu adressieren. Dabei legt sie ein besonderes Augenmerk auf die räumliche Dimension von Informationen. Das Feld umfasst die Sammlung oder Beschaffung, Modellierung, Verarbeitung, intensive Analyse sowie die Darstellung und Verbreitung von geografischen Daten (Lange, 2020, S. 4). Die Nutzung von Künstlicher Intelligenz (K.I) in der Geoinformatik blickt auf eine lange, wenn auch ungleichmäßige Entwicklung zurück. Schon in den späten 1980er Jahren begannen Forschende damit, grundlegende K.I-Techniken wie regelbasierte Expertensysteme auf geographische Fragestellungen anzuwenden (Peter F. Fisher, 1989, S. 279) Die Verbindung zwischen Geografie und Künstlicher Intelligenz (K.I) reicht bis in die 1990er Jahre zurück. Schon vor den bedeutenden Fortschritten im Bereich der GeoAI integrierte man zahlreiche K.I-Methoden in die Geoinformatik und entwickelte sie weiter. Bereits in den 1980er Jahren nutzte man künstliche neuronale Netze (ANN), heuristische Suchverfahren, wissensbasierte Expertensysteme, Neuroinformatik sowie künstliches Leben wie zelluläre Automaten. In den 1990er Jahren folgten dann neuartige Entwicklungen wie die sogenannte genetische Programmierung, Fuzzy-Logik und hybride intelligente Systeme.

In den 2000er Jahren kamen Ontologien, Websemantik und geografisches Information Retrieval (GIR) hinzu. Seit etwa 2010 hat das Deep Learning eine bemerkenswerte Entwicklung erlebt, insbesondere durch Erfolge bei der Ausbildung tiefer neuronaler Netzwerke (DNNs). Ein bedeutender Meilenstein war im Jahr 2012 die Leistung von AlexNet. Dieses tiefe neuronale Netzwerk beeindruckte bei der ImageNet Large Scale Visual Recognition Challenge (Gao, Hu & Li, 2024, S. 4)

Ein junger Forschungszweig, der Künstliche Intelligenz und Räumliche Fragestellungen verbindet ist die Geospatial Artificial Intelligence (GeoAI) wichtig ist hier anzumerken, dass sich das Feld der GeoAI nicht nur auf die Anwendung von ML oder DL-Techniken auf räumliche Fragestellungen beschränkt, sondern zudem räumliche, zeitliche und ortsbezogene Aspekte in Methoden der K.I. integrieren. So wird zum einen die Transdisziplinarität gefördert und zum anderen vermieden, dass sich die Forschung in spezialisierte, eng gefasste Bereiche aufspaltet, die wenig miteinander kommunizieren (Gao et al., 2024, S. 27)

2.2.1 Grundlagen der Geoinformationssysteme

Das Geoinformationssystem „GIS“ dient als Werkzeug in der Geoinformatik. Ein Informationssystem umfasst die Aufnahme, Speicherung, Aktualisierung, Verarbeitung und Auswertung von Informationen sowie deren Wiedergabe. Geoinformationssysteme bieten zusätzlich die Besonderheit, dass Geoobjekte Geometrie und Topologie als implizite Bestandteile enthalten. Die Verarbeitung solcher raumbezogener Informationen benötigt spezielle Werkzeuge oder Funktionen. Diese werden von anderen Informationssystemen nicht bereitgestellt (Lange, 2020, S. 373).

Es folgt eine kurze Geoinformationssystem Definition

> *„Ein Geoinformationssystem dient der Erfassung, Speicherung, Analyse und Darstellung aller Daten, die einen Teil der Erdoberfläche und die darauf befindlichen technischen und administrativen Einrichtungen sowie geowissenschaftliche, ökonomische und ökologische Gegebenheiten beschreiben (Bartelme, 2005, S. 13)“*

Die Entwicklung von Geoinformationssystemen (GIS) startete in den 1960er Jahren. Eines der frühesten Systeme war das "Canada Geographic Information System" (CGIS). Roger Tomlinson, bekannt als "Vater von GIS", entwickelte es zur Analyse umfangreicher ländlicher Daten Kanadas (Canada Land Inventory, CLI). Später kamen innovative Ansätze aus dem Harvard Laboratory For Computer Graphics And Spatial Analysis. Entwicklungen wie Arc/Info von ESRI

und das rasterbasierte GRASS GIS vom US Army Corps of Engineers prägten diese Zeit. In den 1990er Jahren beschleunigten erhebliche Hardware- und Software-Verbesserungen die Fortschritte. Heute sind GIS-Tools Standard für Kommunen, Planungsbehörden sowie Verkehrs- und Telekommunikationsunternehmen.

2.2.1.1 Geoinformationssystem-Software

In dieser Arbeit wird die freie Geoinformationssoftware QGIS verwendet. Bei QGIS handelt es sich um ein Geoinformationssystem zum Betrachten, Transformieren, Erfassen und Analysieren von Geodaten und ist unter der GNU General Public License lizenziert. Es läuft unter den gängigen Betriebssystemen: Linux, Unix, Mac, OSX, Windows und Android. Außerdem unterstützt es mehrere Vektor- Raster- und andere Datenformate (QGIS Österreich, 2021) Gegenüber proprietären Lösungen wie ArcGIS von ESRI zeichnet sich QGIS durch folgende Aspekte aus: QGIS ist frei verfügbar, damit unabhängig von kommerziellen Lizenzmodellen einsetzbar und besonders für Forschungsprojekte mit begrenztem Budget attraktiv. Darüber hinaus fördert der Open-Source-Charakter die Transparenz und Anpassbarkeit der Software an spezifische Anforderungen. Eine Vielzahl von Plug-ins ermöglicht die Erweiterung der Grundfunktionalität (Lange, 2020, S. 379), z.B. durch die Integration von Python-Skripts, die im Rahmen dieser Arbeit zur Automatisierung von Geodatenverarbeitungsprozessen eingesetzt wurden. Proprietäre Systeme wie das weltweit häufig verwendete ArcGIS bieten zwar häufig eine tiefere Integration in größere Software-Ökosysteme und spezifische Funktionalitäten wie ArcGIS Online für cloudbasierte Anwendungen (Lange, 2020, S. 378), sind jedoch kostenintensiv und in ihrer Anpassbarkeit eingeschränkt. In dieser Arbeit wurde bewusst eine offene und flexible Lösung gewählt, um die Reproduzierbarkeit und Zugänglichkeit der Ergebnisse zu verbessern. Der Hauptgrund, warum sich in dieser Arbeit für QGIS entschieden wurde, liegt daran, dass durch die QGIS Python Konsole eine einfache Schnittstelle besteht, in der man den von LLM`s generierten Code ausführen kann und somit geeignet für die geplanten Tests ist. In der folgenden Tabelle 2 wird ein Überblick über die relevanten GIS-Softwareprogramme gegeben.

	Software	Beschreibung
1	ArcGIS	Ein weit verbreitetes GIS-System von ESRI mit cloudbasierten Angeboten wie ArcGIS Online.
2	GeoMedia	Ein flexibles GIS-Managementsystem, das von Hexagon Geospatial angeboten wird.
3	MapInfo Pro	Desktop-GIS-Software mit umfangreichen Analysemöglichkeiten durch MapInfo Vertical Mapper.
4	Smallworld	Ein GIS-System von General Electric, das vor allem in der Energie- und Wasserwirtschaft genutzt wird.
5	GRASS GIS	Ein Open-Source-GIS-System, das von Regierungsbehörden und akademischen Einrichtungen genutzt wird.
6	gvSIG	Eine plattformunabhängige Open-Source-GIS-Software mit Desktop-, Online- und Mobile-Versionen.
7	OpenJUMP	Ein in Java geschriebenes Open-Source-GIS, das auf Vektordaten spezialisiert ist.
8	QGIS	Ein leistungsstarkes Open-Source-GIS mit umfangreichen Dokumentationen und einer Vielzahl von Plugins.
9	Spring GIS	Ein modernes GIS- und Bildverarbeitungssystem aus Brasilien mit Integration von Raster- und Vektordaten.
10	SAGA	Ein an der Universität Göttingen entwickeltes GIS, spezialisiert auf hydrologische Modellierung und Geostatistik.

Tabelle 2: Relevante GIS-Software im Überblick (Lange, 2020, S. 378–380) Eigene Darstellung 2025

2.2.1.2 Programmieren und Programmiersprachen in Geoinformationssystemen

Die Nutzung von Programmierung in Geoinformationssystemen bietet den Vorteil, dass diese Softwareprogramme dadurch individualisiert und erweitert werden können, um maßgeschneiderte Lösungen für spezifische geographische und raumbezogene Fragestellungen bereitzustellen. Durch den Einsatz von Skriptsprachen wie Python oder JavaScript, die häufig in GIS-Umgebungen wie QGIS oder ArcGIS integriert sind, können Benutzer komplexe Automatisierungen und Analysen effizient umsetzen. Dies umfasst beispielsweise die Automatisierung repetitiver Aufgaben, die Integration externer Datenquellen oder die Entwicklung benutzerdefinierter Visualisierungswerkzeuge. Darüber hinaus ermöglicht die Programmierung in GIS die Entwicklung innovativer Analyseverfahren, die über die standardmäßig verfügbaren Funktionen hinausgehen. Benutzer können beispielsweise Algorithmen zur räumlichen Optimierung implementieren, neue Geodatenformate verarbeiten oder Workflows für komplexe räumliche Analysen automatisieren. Mit der Möglichkeit, externe Bibliotheken und Frameworks wie GeoPandas, Shapely oder Rasterio zu integrieren, erweitert sich das Spektrum der möglichen Anwendungen erheblich. Ein weiterer Vorteil der Programmierung liegt in der Möglichkeit, Prozesse besser nachvollziehbar und reproduzierbar zu machen. Dies ist insbesondere in wissenschaftlichen oder groß angelegten industriellen Projekten von Bedeutung, wo die Dokumentation und Wiederholbarkeit von Analysen entscheidend sind. Durch die Verwendung von Versionskontrollsystemen wie GIT können Änderungen am Code überwacht und historische Analysen effizient nachvollzogen werden. Die programmatische Erweiterbarkeit ermöglicht auch die Integration von KI-gestützten Algorithmen, beispielsweise für die Klassifikation von Satellitendaten oder für die Vorhersage von Umweltveränderungen. Solche Ansätze kombinieren die traditionelle GIS-Funktionalität mit modernen maschinellen Lernverfahren, was die Anwendungsfelder von GIS erheblich erweitert (Lange, 2020, S. 74–75), (*What are the Top Programming Languages in the GIS World? – GoGeomatics,* 2025o). Neben anderen beliebten Programmiersprachen wie C++ oder R, die im Kontext von GIS genutzt werden, kommt Python zur Automatisierung von GIS-Aufgaben eine bedeutende Rolle zu (Lange, 2020, S. 74) und ist in den führenden Systemen wie ArcGIS (proprietär) und QGIS (frei verfügbar) verwendet (*What are the Top Programming Languages in the GIS World? – GoGeomatics,* 2025o). QGIS erlaubt es darüber hinaus, dem Benutzer eigene Funktions-Erweiterungen (Plug-Ins) mit Python zu schreiben, die benutzerdefinierte Funktionen aufweisen (Lange, 2020, S. 74). Die Wahl von der Programmiersprache Python für meine Masterarbeit begründet sich durch ihre weit verbreitete Nutzung und Vielseitigkeit durch Integration von verschiedenen GIS-Bibliotheken (GISGeography, 2022). In der folgenden Tabelle 3 findet sich ein kurzer Überblick der relevantesten Programmiersprachen im GIS-Bereich.

Programmiersprache	Details
Python	Python ist vielseitig mit vielen Bibliotheken. Es ist die erste Wahl für Automatisierung, Skripterstellung und Integration mit verschiedenen GIS-Bibliotheken.
R	R eignet sich hervorragend für statistische Analysen und die Visualisierung von räumlichen Daten.
SQL	Räumliche Datenbanken basieren stark auf SQL für grundlegende Operationen. Dadurch ist es ein unverzichtbares Werkzeug für Abfragen geografischer Daten.
JavaScript	JavaScript ist essenziell für webbasierte GIS-Anwendungen und interaktive Karten. Schauen Sie sich Beispiele der Esri JavaScript API für Webmaps und Szenen an.
Java	Java wird hauptsächlich für die Entwicklung von Desktop-GIS-Anwendungen verwendet, einschließlich GIS-Bibliotheken und APIs wie GeoTools.
C++	C++ ist eine bevorzugte Wahl für Desktop-GIS-Software und leistungsstarke Anwendungen. Es ist sozusagen die geheime Zutat hinter den Kulissen von QGIS 3.

Tabelle 3: Überblick der relevanten Programmiersprachen im GIS-Kontext (GISGeography, 2022) Eigene Darstellung 2025

3 Methodik

In dieser Masterarbeit wird die Fähigkeit von Large Language Models (LLM's) zur Generierung von Python-Code für Geodatenverarbeitungsaufgaben umfassend evaluiert. Zur Beantwortung der Forschungsfragen sollen die im Literaturverzeichnis aufgeführten Artikel in einer vergleichenden Literaturrecherche analysiert werden, um das nötige Basiswissen zu erschließen. Danach folgt die Auswahl der LLMs, die nach definierten Kriterien wie Benchmarks (insbesondere in der Fähigkeit Code zu generieren), Verfügbarkeit (proprietär oder Open Source) und Datenbasis (Code LLM, Foundation Model) erfolgt. Anschließend werden spezifische Geodatenverarbeitungs-Aufgaben definiert, die typische Szenarien in der Geoinformationsverarbeitung darstellen. Beispielsweise: Automatisierte Feature-Erstellung und Manipulation, Raumanalysen (Spatial Analysis) wie das Erzeugen von Pufferzonen um Features und die Automatisierung eines mehrstufigen Geodatenverarbeitungs-Workflows. Für die Evaluierung wird das HumanEval Benchmark (Chen et al., 2021) herangezogen, insbesondere der Pass@k-Metrik, die definierte Kriterien für die Bewertung der generierten Codes verwendet z.B. syntaktische Korrektheit (Die Anzahl der fehlerfreien Ausführungen, der kompiliert, ohne Syntaxfehler auszuführen ist), Accuracy (Genauigkeit) Anteil der erfolgreichen Codeausführungen im Verhältnis zu gescheiterten Versuchen. Außerdem die Erfüllung funktionaler Anforderungen (Komplette Lösung der Testszenarien). Bei der angewendeten Prompting-Technik wurde sich dafür entschieden die Experimente auf Zero-Shot zu beschränken, wodurch ein mehr realistischer Anwendungsfall simuliert wird, da es im praktischen Einsatz oft nicht realistisch ist, dass Nutzer für jede Aufgabe Beispiele, wie beim Few-Shot-Prompting, bereitstellen, da dies oftmals Fachwissen voraussetzt, was nicht immer gegeben ist. Auf Basis dieser Metriken werden Testprotokolle entwickelt und es folgt die Testphase, in der die Experimente in einer kontrollierten Umgebung durchgeführt werden, wobei der von den LLMs generierte Code ausgeführt, protokolliert und anhand der vorab definierten Kriterien analysiert wird. Die gesammelten Daten aus diesen Tests bieten die Grundlage für eine quantitative Bewertung der Leistungsfähigkeit jedes Modells. Die Analyse und der Vergleich der Modelle erfolgen durch die Auswertung der Ergebnisse, um Unterschiede in der Leistung zu identifizieren und zu verstehen, welche Modelleigenschaften zu einer besseren oder schlechteren Leistung führen. Weiterhin werden die Limitationen und Fehler der Modelle untersucht. Abschließend wird der gesamte Prozess detailliert dokumentiert und im Kapitel Ergebnisse zusammengefasst, das die Forschungsergebnisse präsentiert. Im Kapitel Ausblick finden sich dann Empfehlungen für zukünftige Forschungsarbeiten. Diese methodische Herangehensweise gewährleistet eine fundierte Untersuchung der Codegenerierungsfähigkeiten der LLMs und ermöglicht es, substanzielle Schlussfolgerungen über ihre praktische Anwendbarkeit und Verbesserungsmöglichkeiten zu ziehen.

3.1 Auswahl der Large Language Models

In diesem Kapitel erfolgt die systematische Auswahl der LLMs, die in den nachfolgenden Tests untersucht werden sollen. Die Auswahl basiert auf definierten Kriterien wie den verfügbaren Benchmarks, insbesondere der Fähigkeit Code generieren zu können, der Zugänglichkeit der Modelle (proprietär vs. Open Source) und Bedienbarkeit (User Interface). Ein zentraler Aspekt bei der Auswahl der LLM's ist die zuverlässige Bewertung ihrer Leistungsfähigkeit. Diese Benchmarks ermöglichen es, LLMs nach Kriterien wie Genauigkeit, Effizienz und Vielseitigkeit zu bewerten. Die Berücksichtigung dieser Kriterien im Evaluationsprozess stellt sicher, dass im Vergleich nur die bestgeeigneten Modelle in Betracht gezogen werden und bietet somit eine fundierte Basis für die Auswahl der LLMs.

3.1.1 Auswahlkriterien

Bei der Auswahl der Large Language Models gibt es verschiedene Kriterien, die zur Evaluierung zu Rate gezogen werden können:

Benchmarking:

Es gibt offizielle Benchmarks die einen Indikator über die generelle Leistungsfähigkeit der Modelle. Hierfür werden in dieser Arbeit Benchmarks von Hugging Face zu Rate gezogen. Hierbei handelt es sich um ein führendes Unternehmen im Bereich der künstlichen Intelligenz, das sich auf die Bereitstellung von Modellen und Tools für die Verarbeitung natürlicher Sprache (Natural Language Processing, NLP) spezialisiert hat. Das Unternehmen hat eine offene und benutzerfreundliche Plattform entwickelt, die es Forschern und Entwicklern ermöglicht, LLMs zu teilen, zu bewerten und zu verwenden. Die Plattform bietet ein breites Spektrum an Modellen, sowie zugehörige Datensätze, um eine Vielfalt von NLP-Aufgaben zu bewältigen. Ein wesentlicher Teil des Angebots von Hugging Face sind die sogenannten Benchmarks, standardisierte Tests, die dazu dienen, die Leistung verschiedener Modelle vergleichbar zu machen. (The Big Benchmarks Collection - a open-llm-leaderboard Collection, 2025b) Da die Leistungsfähigkeit ein Hauptkriterium bildet, wenn es darum geht komplexe Geodatenverarbeitungsaufgaben zu lösen wurde sich in dieser Arbeit für das Model GPT o1-preview entschieden, da dieses einen neuartigen Ansatz zur besseren Lösung von Aufgaben verwendet. In dieser Arbeit wurde außerdem Claude 3.5 Sonett von Anthropic ausgewählt, da es in einem allgemeinen Benchmark von Hugging Face einen hohen Score erreicht (Chatbot Arena Leaderboard - a Hugging Face Space by lmarena-ai, 2024a)

Datenbasis:

Des Weiteren wurde bewusst ein Modell ausgewählt, das spezifisch auf Basis von Code trainiert wurde. In Abbildung 8 sehen wir einen Benchmark von Hugging Face für verschiedene Modelle, die explizit für Codegenerierung trainiert wurden. In dieser Arbeit wurde sich für das open source Modell Deepseek Coder entschieden, da dieses sowohl einen hohen Benchmark-Score im Bereich Programmieren erreicht, als auch eine problemlose Bedienbarkeit über ein User-Interface ermöglicht.

Transparenz/Ethik:

Dies ist ein weiterer Grund warum gegenüber anderen führenden LLM's das Model Claude von Anthropic ausgewählt wurde, weil es den Fokus auf ethischen Umgang gelegt hat, indem es auf einer vorgegebenen Liste von Regeln und Prinzipien trainiert wurde um potenziell schädliches Verhalten, was den Nutzern oder der Gesellschaft Schaden zufügen könnte, zu vermeiden (Bai et al., 2022, S. 5).

Bedienbarkeit/Zugänglichkeit:

Es bestehen teilweise Unterschiede, wie auf das Modell zugegriffen werden kann, also über welchen Weg ein Eingabe-Prompt dem Model übergeben wird. Die meisten proprietären Modelle bieten ein User-Interface, während Modelle wie Metas Opensource-Modell „Llama" nur über Code-Eingabe in der Kommandozeile und je nach Parametergrößen-Version über den lokalen Rechner laufen muss und damit auch massive Performance-Anforderungen erfordert. Da meine Performance-Kapazitäten begrenzt sind, werden diese Modelle nicht berücksichtigt. Stattdessen wurde sich für das Opensource-Modell „Deepseek Coder" mit komfortabler UI entschieden.

3.1.2 Technische Beschreibung der ausgewählten Modelle

3.1.2.1 Deepseek Coder

Dieses Open Source Modell eines chinesischen Unternehmens wurde speziell für Codegenerierung und -Verständnis als frei zugängliche Alternative zu proprietären Modellen entwickelt. Es wurde auf einem umfangreichen Datensatz von ca. 2 Billionen Token trainiert, welcher 87 verschiedene Programmiersprachen beinhaltet und umfasst in verschiedenen Varianten 1.3 bis 33 Milliarden Parameter. Mit einer Kontextlänge von 16.000 Token kann das Modell große Codeabschnitte verarbeiten (Guo et al., 2024, S. 2). In der folgenden Abbildung 5 wird die Performance von verschiedenen codebasierten Modellen in verschiedenen populären Programmiersprachen verglichen. Hierbei wird deutlich, dass sich Deepseek Coder gegenüber anderen Modellen wie CodeLlama oder Starcoder deutlich abheben kann.

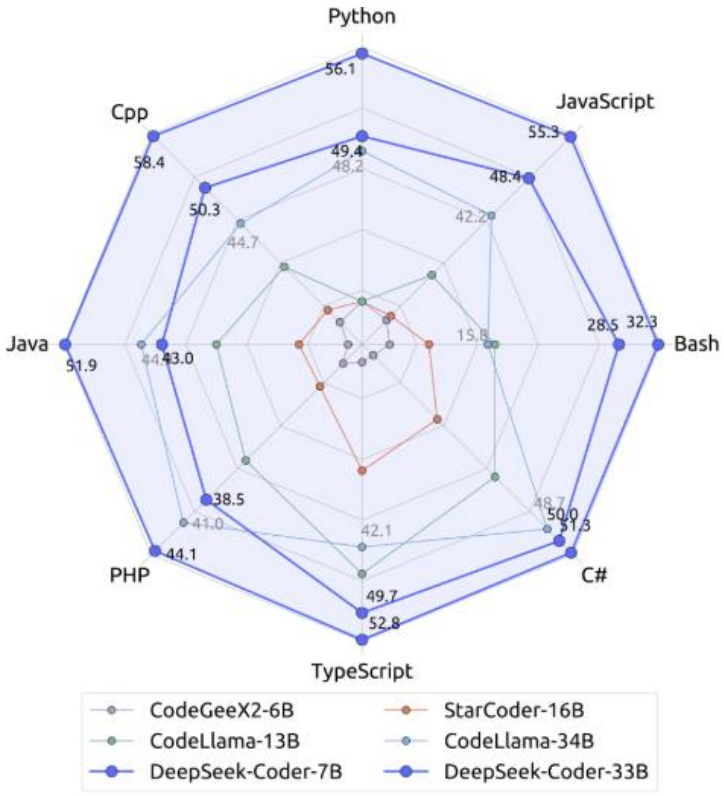

Abbildung 5: Vergleich von Deepseek Coder und anderen codebasierten LLM's in ihrer Performence über verschiedene Programmiersprachen hinweg in multiplen Benchmarks (Guo et al., 2024)

3.1.2.2 Claude 3.5 Sonett

Claude 3.5 Sonnet ist Teil der Claude 3-Modellfamilie, die von Anthropic entwickelt wurde und multimodale K.I-Funktionen bietet. Claude Sonnet ist das mittlere Modell dieser Familie und bietet eine Balance aus Geschwindigkeit und Leistungsfähigkeit, insbesondere in den Bereichen Textverständnis, Codierung und räumliches Denken. (*The Claude 3 Model Family: Opus, Sonnet, Haiku,* 2024b, S. 1) Es bietet ein umfangreiches Kontextfenster mit 200.000 Token und einen maximalen Output von 8192 Token, wie auch aus der Abbildung 6 zu entnehmen ist (2024a, 2024b)

	Claude 3.5 Sonnet	Claude 3.5 Haiku	Claude 3 Opus	Claude 3 Sonnet	Claude 3 Haiku
Description	Our most intelligent model	Our fastest model	Powerful model for highly complex tasks	Balance of intelligence and speed	Fastest and most compact model for near-instant responsiveness
Strengths	Highest level of intelligence and capability	Intelligence at blazing speeds	Top-level intelligence, fluency, and understanding	Strong utility, balanced for scaled deployments	Quick and accurate targeted performance
Multilingual	Yes	Yes	Yes	Yes	Yes
Vision	Yes	No	Yes	Yes	Yes
Message Batches API	Yes	Yes	Yes	No	Yes
API model name	Upgraded version: `claude-3-5-sonnet-20241022` Previous version: `claude-3-5-sonnet-20240620`	`claude-3-5-haiku-20241022`	`claude-3-opus-20240229`	`claude-3-sonnet-20240229`	`claude-3-haiku-20240307`
Comparative latency	Fast	Fastest	Moderately fast	Fast	Fastest
Context window	200K	200K	200K	200K	200K
Max output	8192 tokens	8192 tokens	4096 tokens	4096 tokens	4096 tokens
Cost (Input / Output per	$3.00 / $15.00	$1.00 / $5.00	$15.00 / $75.00	$3.00 / $15.00	$0.25 / $1.25

Abbildung 6: Spezifikationen von Anthropic Claude Large Language (The Claude 3 Model Family: Opus, Sonnet, Haiku, 2024b)

(*The Claude 3 Model Family: Opus, Sonnet, Haiku,* 2024b)

3.1.2.3 GPT o1-preview

Das Sprachmodell o1-preview von Open AI ist eine neue Entwicklung im Bereich der LLM's, das speziell für Planungsaufgaben optimiert wurde. Gegenüber anderen LLMs ist es so konzipiert, dass es Techniken wie Chain-of-Thought (CoT) und Tree-of-Thought (ToT) verwendet. Durch die Einbindung von ToT und CoT in „o1-preview" kann das Modell bei komplexen Planungsaufgaben verschiedene Lösungspfade gleichzeitig durchdenken und so bessere Entscheidungen treffen. Insbesondere, wenn Unsicherheiten oder mehrere mögliche Lösungswege existieren. Der ToT-Ansatz erlaubt es dem Modell, mehrere Gedankengänge, also verschiedene potenzielle Lösungsschritte, gleichzeitig zu entwickeln und zu bewerten. Dies ermöglicht es, Alternativen zu prüfen, zurückzuspringen und zu anderen Wegen zu wechseln, wenn sich ein Pfad als unproduktiv herausstellt.

Außerdem wird die Fähigkeit des Modells gestärkt, mehrschrittige Problemlösungen zu planen und dabei jedes Teilziel konsistent zu verfolgen. Dies ist besonders wichtig in Umgebungen, in denen eine falsche Sequenz von Aktionen das Endziel gefährden könnte, da die Techniken dem Modell helfen, die Problemlösung schrittweise und nachvollziehbar zu gestalten. (Yao et al., 2023, S. 2–3), (Wei et al., 2022, S. 3)

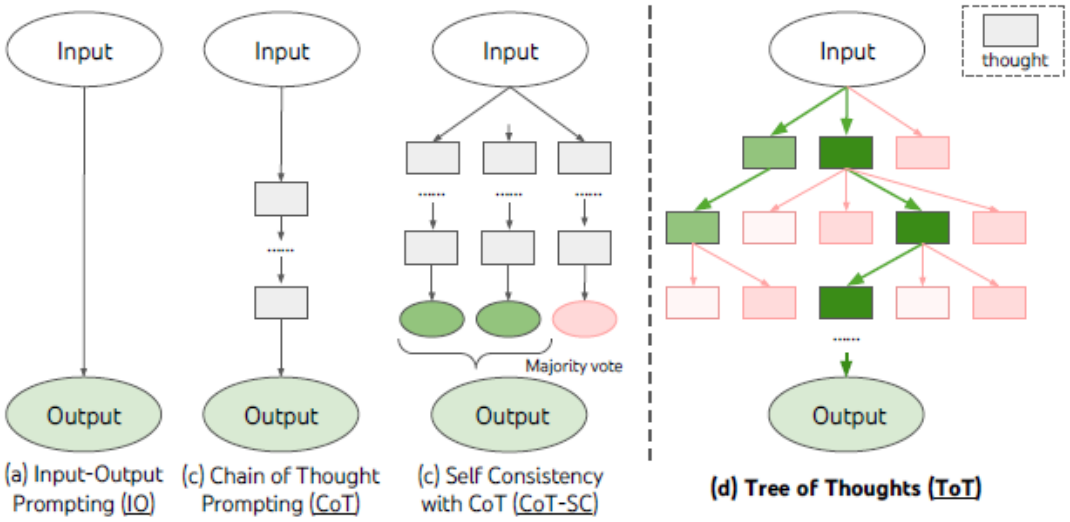

Abbildung 7: Funktionsweise der Tree of Thoughts- und Chain of Thoughts-Technik (Yao et al., 2023, S. 2)

3.2 Auswahl eines Workflows für das Testszenario

Um die ausgewählten Modelle auf ihre Leistungsfähigkeit zu überprüfen, wurden drei verschiede Testszenarien erarbeitet, welche reale Anwendungsfälle in der Geoinformationsverarbeitung darstellen könnten. Da die Energiebranche in Zeiten der Energiewende eine wichtige Rolle spielt, wurde sich dafür entschieden Daten von Gas-, Biogas- und Wasserstoffleitungen in den Szenarien zu verwenden. Da in diesem Kontext die Infrastrukturen mit hoher Genauigkeit beschrieben werden müssen, wurde sich für Vektor- gegenüber Rasterdaten entschieden. Um eine hohe Praxisnähe zu gewährleisten wurde sich auf amtliche oder realitätsnahe Quellen bezogen wie die Global Oil & Gas Features Database (Sabbatino, 2018) und Openstreetmap-Daten.

3.2.1 Auswahlkriterien der Testszenarien

Die ausgewählten Testszenarien sollen repräsentativ für klassische Geodatenverarbeitungsaufgaben sein, aus diesem Grund wurde sich für räumliche Vektordaten-Analysen mit Geoprozessierungs-Werkzeugen wie Puffer und Verschneidung inklusive des Umgangs mit Koordinaten-Referenzsystemen entschieden. Diese Operationen gehören zu den klassischen Vorgehensweisen in der Geoinformatik (Lange, 2020, S. 401) Des Weiteren, wurde darauf Wert gelegt, dass sich die Szenarien in ihrer Komplexitätsanforderung untereinander unterscheiden, um festzustellen, wie leistungsfähig die Modelle bis zu welchem Grad sind und wann sie an ihre Grenzen stoßen. Diese Vorgehensweise ist besonders herausfordernd für LLMs, da sie sowohl syntaktische Programmierkenntnisse als auch semantisches Verständnis von Geodaten und deren Verarbeitung erfordern.

In der **1. Komplexitätsstufe** (Testszenario 1) werden wenige Verarbeitungsschritte und keine klassischen Geodatenverarbeitungs-Werkzeuge gefordert.

Auf der **2. Komplexitätsstufe** (Testszenario 2) werden Werkzeuge wie Buffer und Intersect vom Modell gefordert, aber die Anforderung an logisches Schlussfolgern und die Anzahl der Verarbeitungsschritte bleibt gering.

In der **3. Komplexitätsstufe** (Testszenario 3) kommt hinzu, dass eine Längenberechnung notwendig ist, die eine Koordinatensystem-Transformation oder Ellipsoid-Berücksichtigung erfordert. Außerdem verlangt es die größte Anzahl an verschiedenen Verarbeitungsschritten der Szenarien. Für die jeweiligen Untersuchungsgebiete wurde sich auf kleine lokale Gebiete

beschränkt, um die zur Verfügung stehenden Rechnerkapazitäten und die Kapazitäten von QGIS nicht zu überschreiten, daher bilden die Ausgangsdaten 20 bis maximal 400 Features. Ein weiteres Kriterium, worauf bei der Auswahl der Testszenarien wert gelegt wurde, ist das verschiedene Arten von Koordinatenreferenzsystemen verwendet werden. Eines, welches geografische Koordinaten und Gradeinheiten verwendet, gegenüber welchen, die sich bestimmte Regionen wie Mitteleuropa oder die USA eignen und Meter oder Foot als Maßeinheit verwenden. So kann festgestellt werden, ob das Modell nötige Transformationen und Einheitenumrechnungen anwendet und mit unterschiedlichen Koordinatensystemen umgehen kann.

3.2.2 Beschreibung Testszenario 1: Manipulation Gasleitungs-Features in QGIS

Im **Szenario 1** werden Geometrien von Gasleitungen Österreichs, aus der Global Oil & Gas Features Database (Sabbatino, 2018) verwendet. Dieser Datensatz enthält einen Layer mit Hoch und Mitteldruck-Gasleitungen in Form von Line-Features, es soll nun per Python-Skript für die QGIS Python-Konsole ein zusätzlicher Layer erstellt werden, mit Polygon-Features, der Gasstationen symbolisiert. Diese Features sollen aus Vierecken mit 50 Meter Seitenlänge bestehen und an den Endpunkten der Gasleitungen lokalisiert werden. Des Weiteren sollen solche Leitungen, die in den Attributen ein Biomethanpotential von über 45 % aufweisen, grün eingefärbt werden, mit samt den Stationen, die an den Endpunkten dieser Leitungen liegen. Das Koordinatensystem für die neu erstellten Layer soll EPSG: 32633 sein. Dieses Szenario wurde ausgewählt, weil die Aufgabe, geometrische Daten zu manipulieren und diese in ein spezifisches Koordinatensystem zu transformieren, eine komplexe Verarbeitung und Manipulation von Geodaten darstellt. Sie erfordert den Umgang mit räumlichen Datenstrukturen sowie geographischen Projektionen und Transformationen. Es wurde hierbei explizit eine mehrstufige Aufgabe ausgewählt, der Code muss zunächst neue Layer erstellen, dann basierend auf bestimmten Attributwerten Stilregeln anwenden und neue Features erstellen und die Layer gegebenenfalls in das Ziel-Koordinatensystem transformieren. Diese Struktur erlaubt eine detaillierte Analyse der Leistungsfähigkeit des Modells. In Abbildung 8 sind die verschiedenen Schritte für die Umsetzung des Workflows dargestellt.

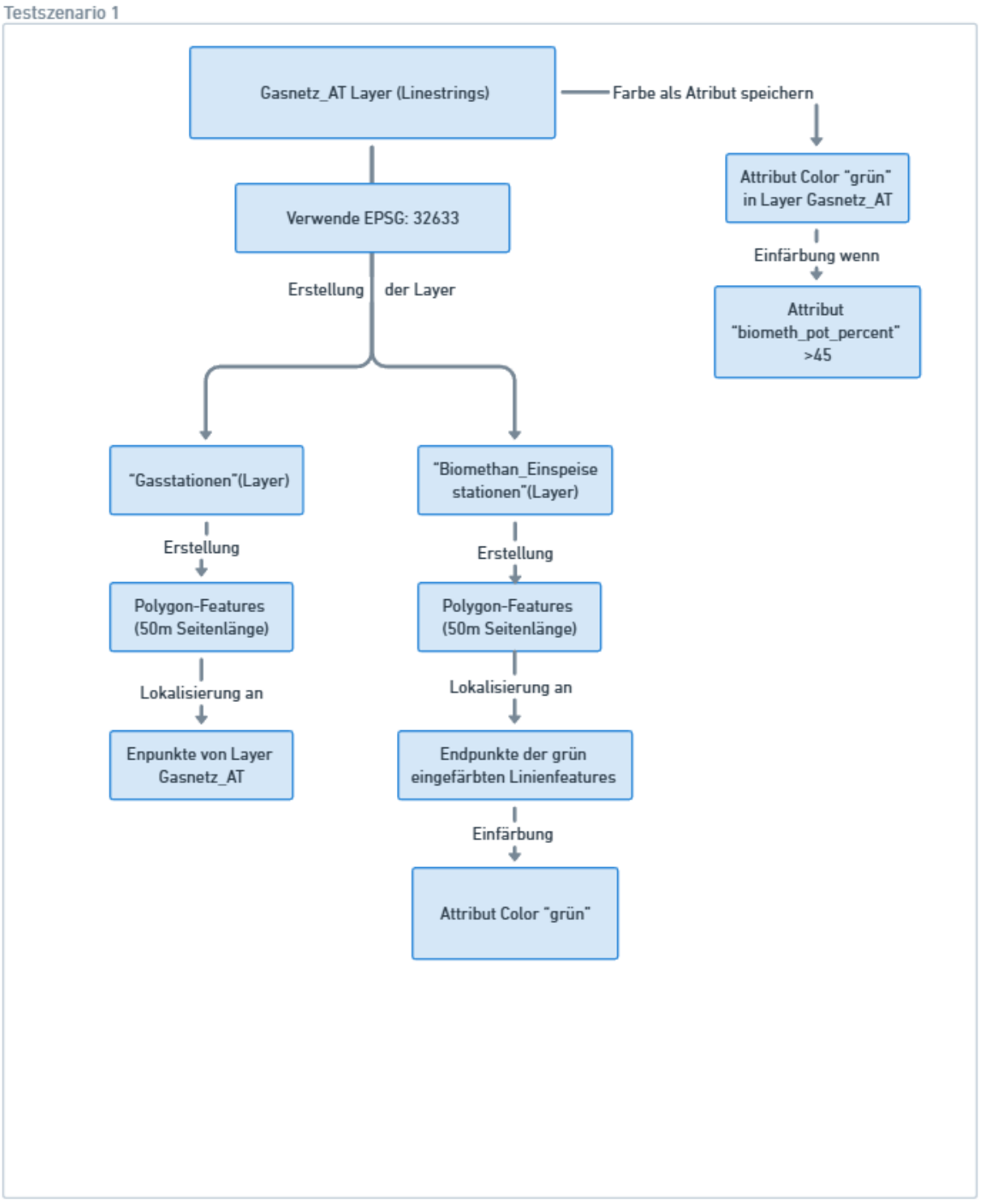

Abbildung 8: Ablaufdiagramm der sequenziellen Schritte für Testszenario 1 (Eigene Darstellung 2024)

3.2.3 Beschreibung Testszenario 2: Gebäude- und Stromversorgungsnetzanalyse

Im **Testszenario 2** besteht die Aufgabe darin, einen von OSM bezogenen Polygon-Layer zu manipulieren. Dieser Layer symbolisiert Gebäude in Klosterneuburg (Österreich), die durch ein Stromversorgungsnetzwerk miteinander verbunden werden sollen. Es darf dabei keine Verbindungsleitungsleitung 500 Meter überschreiten. außerdem soll bei jedem Gebäude mit mehr als 3 Verbindungen ein weiterer Layer mit Polygonen erstellt werden, der Umspannstationen symbolisiert. Da in der Anforderung die Anweisung enthalten ist, das EPSG: 4326 Koordinatensystem zu verwenden, besteht eine Herausforderung dabei, die korrekten Einheiten für die Berechnungen der Entfernungen zu berücksichtigen, da das EPSG: 4326 in Grad misst. Das zweite Szenario weist dadurch die höchste Komplexität und Anforderung auf. In der folgenden Abbildung 9 findet sich ein Workflow-Ablaufdiagramm zu diesem Szenario.

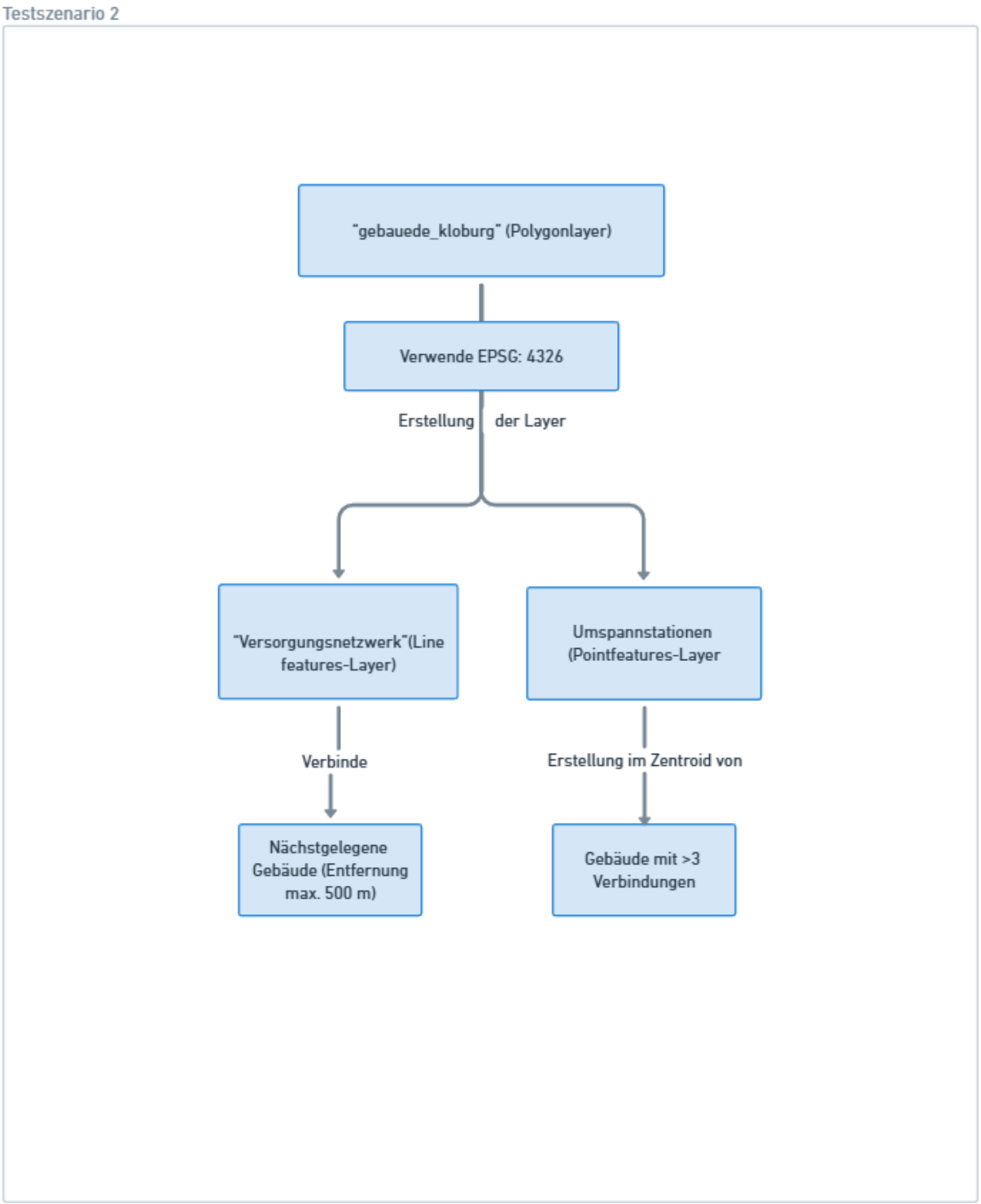

Abbildung 9: Ablaufdiagramm der sequenziellen Schritte für Testszenario 2 (Eigene Darstellung 2024)

3.2.4 Beschreibung Testszenario 3: Wasserstoff-Leitungen und Planung von Hydrolyse-Stationen

Im **3. Szenario** wird ein Datensatz von Wasserstoffleitungen in Houston (USA) von der Innovative Data Energy Applications (IDEA) Group verwendet (Innovative Data Energy Applications, 2025m). Hierbei soll das Modell klassische Geodatenverarbeitungswerkzeuge wie „Intersect“ und „Buffer“ verwenden. In Abbildung 10 findet sich das Ablaufdiagramm dazu.

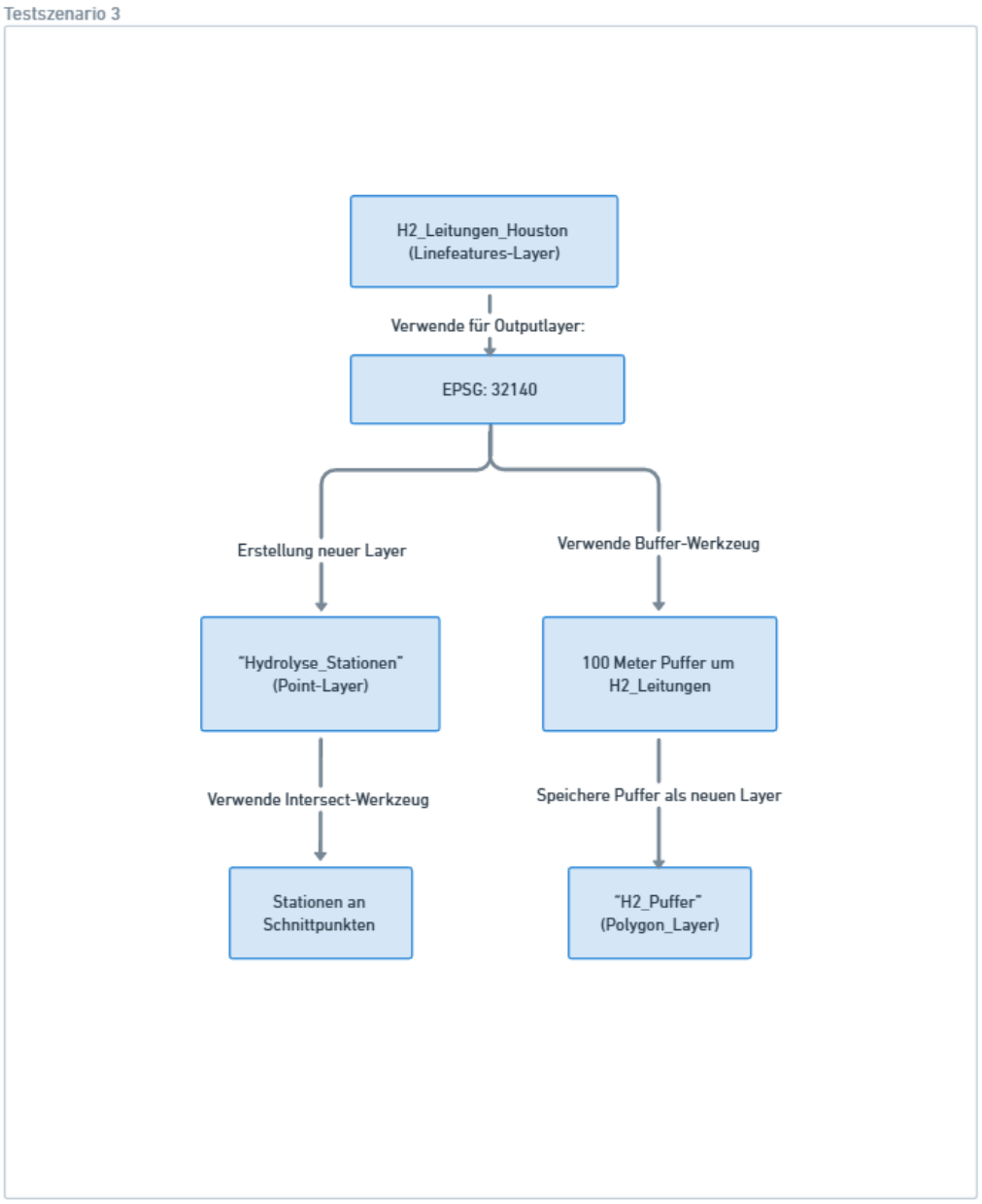

Abbildung 10: Ablaufdiagramm der sequenziellen Schritte für Testszenario 3 (Eigene Darstellung 2024)

3.3 Bewertungs- und Evaluierungskriterien

Generell liegt der Fokus dieser Arbeit darauf, die Codeergebnisse auf funktionale Korrektheit hin zu überprüfen und weniger auf formale Richtigkeit, da die Unterstützung der LLM's bei den Geoverarbeitungs-Workflows vor allem auf Anwender abzielt mit weniger ausgeprägten Programmierungs-Fertigkeiten und dadurch das Ergebnis mehr im Vordergrund steht als die perfekte Anwendung von Programmierung.

3.3.1 Genauigkeit (Accuracy)

Eine simple, aber effektive Bewertungsmethode bietet die Berechnung der „Accuracy" also Genauigkeit der Ergebnisse, indem die Anzahl der korrekten Lösungen durch die Gesamtzahl aller getesteten Aufgaben geteilt wird. Eine Lösung gilt als korrekt, wenn sie alle Anforderungen erfüllt und alle Tests besteht. Das Ergebnis der Berechnung kann einen Wert von 0 bis 1 einnehmen, wobei 1 für eine Genauigkeit von 100 % steht. Die Accuracy liefert somit einen quantitativen Überblick über die Fähigkeit des Modells korrekte Ergebnisse zu generieren. Die Formel dazu findet sich in der folgenden Abbildung 11.

$$\text{Accuracy} = \frac{\text{Anzahl korrekter Lösungen}}{\text{Gesamtanzahl der getesteten Aufgaben}}$$

Abbildung 11: Formel zur Berechnung der Accuracy-Metrik (Eigene Darstellung 2024)

3.3.2 HumanEval Methode

In dieser Arbeit wird als weiteres Bewertungskriterium für den generierten Code die HumanEval Methode herangezogen. Hierbei handelt es sich um eine Bewertungsmethode, die speziell für die Evaluierung von LLMs entwickelt wurde, welche auf Codegenerierung trainiert wurden und sich damit für die angestrebten Tests eignet (Chen et al., 2021, S. 3). Zunächst wird eine spezifische Aufgabenstellung als Prompt für das zu testendes Modell definiert. Im nächsten Schritt wird der Prompt mehrmals, z.B. 50-mal (n = 50) dem Modell übergeben. Anschließend werden sogenannte „Unit Tests" erstellt, die den erstellten Code auf einzelne Funktionen überprüfen. Die Unit Tests werden dann für einen zufällig ausgewählten Teil der generierten Code-Samples ausgeführt, z.B. für 10 generierte Lösungen (k = 10) und überprüft, ob die Lösung der Aufgabe korrekt erfüllt ist. Wenn ein Code-Sample alle Unit-Tests besteht, gilt es als korrekt. Im letzten Schritt wird dann die sogenannte „**Pass@k**" Metrik berechnet, die angibt, wie viele der generierten Samples eine korrekte Lösung darstellen. Die Pass@k-Metrik wird berechnet, indem der Anteil der Aufgaben bestimmt wird, bei denen mindestens eines der k geprüften Samples aller Unit-Tests erfolgreich besteht. Dies ermöglicht eine quantitative

Bewertung der Leistung des Modells, indem gemessen wird, wie oft es in der Lage ist, aus den generierten Lösungsvorschlägen eine funktional korrekte Antwort zu liefern.

Eine höhere Pass@k-Rate (Ergebnis kann einen Wert von 0 bis 1 einnehmen) deutet darauf hin, dass das Modell eine höhere Wahrscheinlichkeit hat, eine korrekte Lösung zu generieren, was auf eine bessere Zuverlässigkeit bei der gestellten Aufgabe hindeutet (Chen et al., 2021, S. 3). In der folgenden Abbildung 12 wird die Formel für die Berechnung der Pass@k-Metrik aufgeführt.

$$\text{Durchschnittlicher Pass@k} = \frac{\text{Summe aller Pass@k-Werte}}{\text{Anzahl der Tasks}}$$

Abbildung 12: Formel zur Berechnung der Pass@k-Metrik (Eigene Darstellung 2024)

In dieser Arbeit wurde sich dazu entschieden, einen moderaten Ansatz zwischen Aussagekraft und Ressourcenaufwand zu verfolgen. Eine zu geringe Anzahl generierter Lösungen reduziert die Aussagekraft der Ergebnisse, da weniger Lösungsvarianten erzeugt werden und damit möglicherweise valide Lösungswege unentdeckt bleiben. Ein zu großes **n** erhöht dagegen den Evaluationsaufwand erheblich und gefährdet damit die Umsetzbarkeit der Forschung. In einer vergleichbaren Studie wurden Werte für **n** zwischen 1 und 100 gewählt (Liu, Xia, Wang & Zhang, 2023, S. 7). In meinen Tests wurde **n** mit 20 und **k** mit 5 festgelegt, was in Anbetracht dessen, dass in der vorher genannten Studie ein komplettes Forschungsteam beteiligt ist und daher die Ressourcen als deutlich höher einzuschätzen sind, vertretbar ist.

3.3.3 BLEU und ROUGE-Metrik

Der **BLEU** (Bilingual Evaluation Understudy)-Score wurde ursprünglich entwickelt, um die Leistung von maschinellen Übersetzungssystemen automatisch zu bewerten. Der BLEU-SCORE vergleicht dabei die maschinell erzeugte Übersetzung mit Referenzübersetzungen, indem er die Übereinstimmungen von n-Grammen (Wortfolgen) zwischen dem Text der Maschine und den Referenztexten misst (Papineni, Roukos, Ward & Zhu, 2001, S. 2). BLEU erzielt dabei hohe Übereinstimmungen mit menschlichen Bewertungen der Übersetzungen (Papineni et al., 2001, S. 9) Auch bei LLM's ist der BLEU-SCORE eine schnelle quantitative Methode zur Bewertung und Vergleich verschiedener Modelle.

Der **ROUGE** (Recall-Oriented Understudy for Gisting Evaluation) -Score bewertet analog zum BLEU die Ähnlichkeit von automatisch generierten Texten und menschlichen Referenzzusammenfassungen, ist aber auf den Recall (Abdeckung) ausgerichtet, das Bedeutet, es wird die Abdeckung des Textes relevanter Inhalte untersucht und weniger die Genauigkeit und Wortwahl. (Chin-Yew Lin, S. 2)

Sowohl die BLEU als auch die ROUGE Metrik wurden ursprünglich für textuelle Aufgaben entwickelt. Es wurde jedoch in Studien gezeigt, dass sie für die Bewertung von Codegenerierung eher weniger gut geeignet sind, da sie nicht die funktionale Korrektheit eines Programms untersuchen können und alternative Metriken wie HumanEval besser geeignet sind (Chen et al., 2021, S. 8). Aus diesem Grund wurde sich in dieser Arbeit gegen die Verwendung von BLEU und ROUGE entschieden.

3.4 Testprotokolle

Testprotokolle bieten eine strukturierte Methode zur Evaluierung der generierten Codes und ermöglichen einen direkten Vergleich der LLM's über die verschiedenen Szenarien hinweg. Durch diese systematische Herangehensweise wird die Reproduzierbarkeit und Validität der Forschungsergebnisse gewährleistet. Die verschiedenen Bereiche im Testprotokoll (Erwartetes Ergebnis, tatsächliches Ergebnis, syntaktische Korrektheit, Ausführbarkeit, Unit Tests, funktionale Anforderungen) wurden so definiert, dass nach dem Ausfüllen die Ergebnisse für die Analysemethoden wie pass@k verwendet werden können. Im Anhang findet sich ein Beispieltestprotokoll.

3.5 Durchführung der Tests

3.5.1 Durchführung Experiment Testszenario 1

Dieses Kapitel beschreibt die Durchführung der Experimente in den verschiedenen Testszenarien. Zunächst wurden die Anforderungen in Form von Prompts zur Lösung der verschiedenen Szenarien definiert und anschließend dem zu testenden Modell übergeben, das Ergebnis wurde dann ohne Korrekturen in die QGIS Python-Konsole eingegeben, um die Ergebnisse zu erhalten. Folgend wird dieser Prozess beschrieben und dokumentiert.

3.5.1.1 Prompt für Testszenario 1:

Erstelle Python-Code für die QGIS Python-Konsole, um die folgenden Schritte auszuführen:

1. Erstellung eines neuen Layers „Gasstationen“:

- Verwende das aktuelle QGIS-Projekt mit dem vorhandenen Layer „Gasnetz_AT“, welcher Line-Features (Gasleitungen) enthält.
- Erstelle einen neuen Layer namens „Gasstationen“, der Polygon-Features repräsentiert.
- Jedes Polygon-Feature soll ein Quadrat mit einer Seitenlänge von 50 Metern sein.
- Diese Vierecke sollen an den Endpunkten der Line-Features im Layer „Gasnetz_AT“ lokalisiert werden.
- Der Layer „Gasstationen“ soll das Koordinatensystem EPSG:32633 verwenden.
- Die Farbe dieser Quadrate soll als Attribut gespeichert werden, z. B. „color“, (damit sie später in der Symbologie angepasst werden kann).

2. Anwenden von Stilregeln auf den Layer „Gasnetz_AT“:

- Prüfe alle Line-Features im Layer „Gasnetz_AT“ und identifiziere diejenigen mit einem Wert größer als 45 in der Attributtabellenspalte „biometh_pot_percent“.
- Erstelle eine Symbologie-Regel, welche diese Line-Features grün einfärbt.
- Die Regel für die Farbgebung sollte als „grün“ in einem neuen Attribut gespeichert werden, z. B. „line_color“, anstatt die Farbe direkt auf das Objekt anzuwenden.

3. Erstellung eines weiteren Layers „Biomethan_Einspeisestationen“:

- Erstelle einen neuen Layer namens „Biomethan_Einspeisestationen“, der Polygon-Features (Quadrate) an den Endpunkten der grün eingefärbten Line-Features aus „Gasnetz_AT“ repräsentiert.

Abbildung 13: Prompt Testszenario 2 in ChatGPT (Eigene Darstellung 2024)

Der in Abbildung 13 dargestellte Prompt wurde anschließend an die ausgewählten Large Language Models übergeben.

In den folgenden Abbildungen 15 und 16 ist ein korrektes Endergebnis visualisiert, welches vom Modell: Deepseek Coder 2.5 mit dem vierten Prompt erstellt wurde. Abbildung 14 zeigt die

unbearbeiteten Gasnetzlinien-Features, nach der Ausführung des generierten Prompts wurde nun ein neuer Layer: „Gasstationen“ im QGIS-Projekt angelegt, der quadratische Polygon Features mit 50 Metern Seitenlänge enthält. Des Weiteren wurde die Symbologie der Line-Features im Layer: „Gasnetz_AT“ angepasst, sodass alle Linien, die in der Attributtabellenspalte: „biometh_pot_percent“ einen Wert größer als 45 enthalten „grün“ eingefärbt wurden.

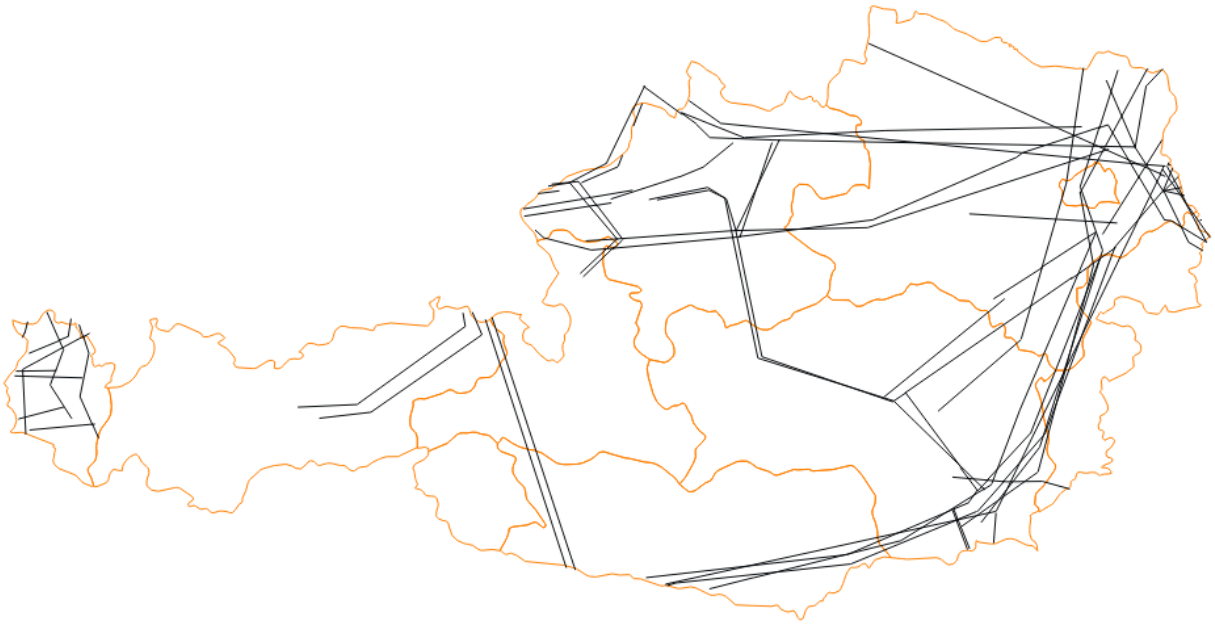

Abbildung 14: Screenshot in QGIS der unbearbeiteten Gasnetzlinien-Features (Eigene Darstellung 2024)

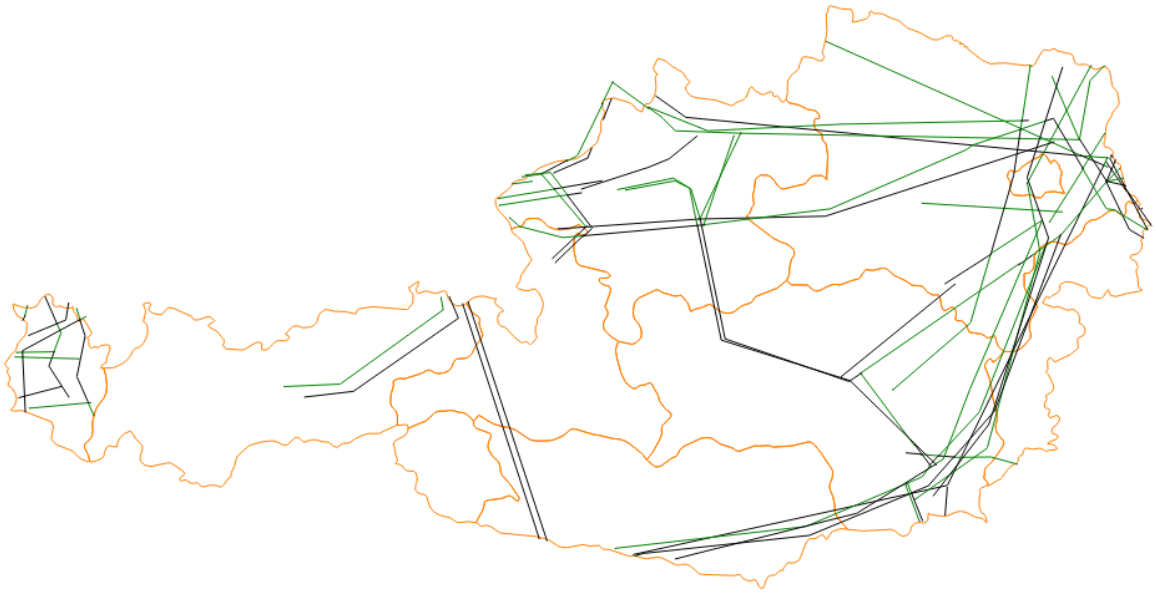

Abbildung 15: Screenshot in QGIS des Testszenario 1 Ergebnis mit grün eingefärbten Features (Eigene Darstellung 2024)

Letztendlich wurde ein weiterer Layer mit dem Namen „Biomethan_Einspeisestationen“ erstellt, der grüne Polygon-Features, die an den Endpunkten der grün eingefärbten Leitungen lokalisiert sind, beinhaltet. Siehe Abbildung 19.

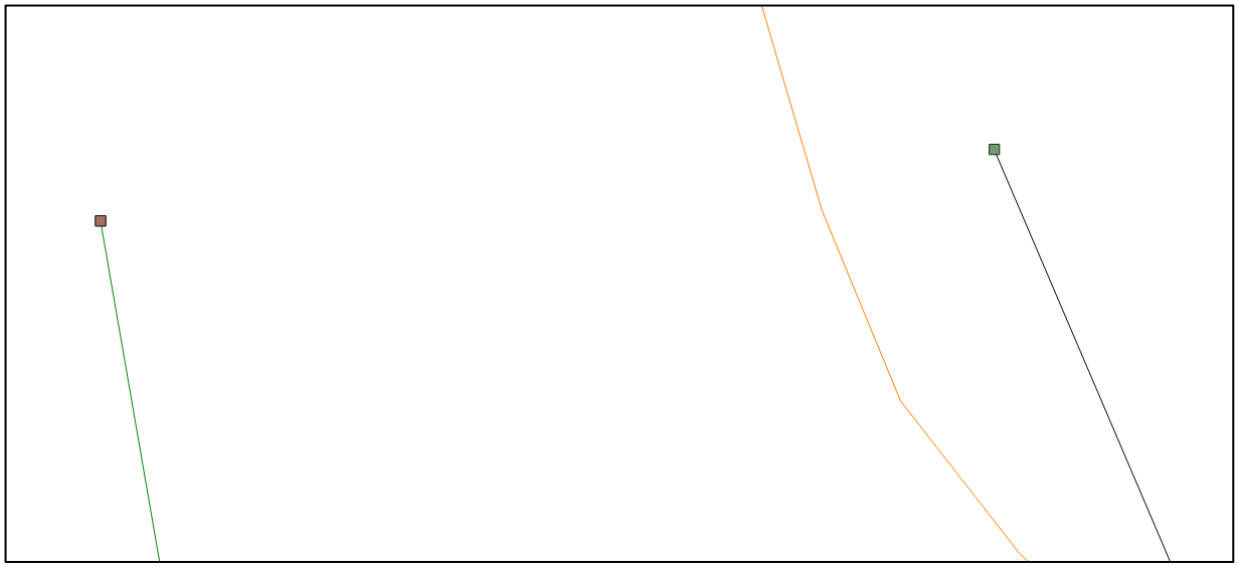

Abbildung 16: Screenshot in QGIS mit erstellten Stationen als Polygon-Layer teilweise grün eingefärbt (Eigene Darstellung 2024)

Die Anforderungen in diesem Testszenario setzen den korrekten Umgang mit Layer Management, Geometriearten wie Punkt- und Line-Features, Koordinatensystemen und die Symbolisierung geografischer Features voraus.

Im folgenden Code ist eine korrekte Lösung für die Aufgabenstellung von Testszenario 1 aufgeführt:

```
# Schritt 1: Erstellung eines neuen Layers "Gasstationen"



# Modulimport: QVariant wird benötigt um später den Datentyp des neuen Attributabellen-Feld
    „color“ als Zeichenkette (String) definieren zu können

from qgis.core import *
from PyQt5.QtCore import QVariant
```

```
# Hier wird auf den im Projekt vorhandenen Layer "Gasnetz_AT" zugegriffen

gasnetz_layer = QgsProject.instance().mapLayersByName('Gasnetz_AT')[0]

# Es folgt die Definition des gewünschten Koordinatensystems EPSG:32633

crs = QgsCoordinateReferenceSystem('EPSG:32633')

# Erstellung eines neuen Polygon-Layers "Gasstationen" im Ziel-Koordinatensystem als
    temporärer Layer

gasstationen_layer = QgsVectorLayer('Polygon?crs=EPSG:32633', 'Gasstationen', 'memory')

# Hinzufügen des neuen Attributs "color" im gasstationen-Layer

gasstationen_layer.dataProvider().addAttributes([QgsField('color', QVariant.String)])
gasstationen_layer.updateFields()

# Funktion zum Erstellen eines Quadrats an einem Punkt

def create_square_at_point(point, side_length):
   half_side = side_length / 2.0
   x = point.x()
   y = point.y()
   square = QgsGeometry.fromPolygonXY([[
      QgsPointXY(x - half_side, y - half_side),
      QgsPointXY(x - half_side, y + half_side),
      QgsPointXY(x + half_side, y + half_side),
      QgsPointXY(x + half_side, y - half_side),
      QgsPointXY(x - half_side, y - half_side)  # Schließen des Polygons
   ]])
   return square

# Iteration über die Features in "Gasnetz_AT" und Erstellung von Quadraten an den Endpunkten

for feature in gasnetz_layer.getFeatures():
   geom = feature.geometry()
   if geom.isMultipart():
```

```
        lines = geom.asMultiPolyline()
        for line in lines:
            start_point = line[0]
            end_point = line[-1]
            for point in [start_point, end_point]:
                square_geom = create_square_at_point(point, 50)
                new_feature = QgsFeature()
                new_feature.setGeometry(square_geom)
                new_feature.setAttributes(['default_color'])
                gasstationen_layer.dataProvider().addFeatures([new_feature])
    else:
        line = geom.asPolyline()
        start_point = line[0]
        end_point = line[-1]
        for point in [start_point, end_point]:
            square_geom = create_square_at_point(point, 50)
            new_feature = QgsFeature()
            new_feature.setGeometry(square_geom)
            new_feature.setAttributes(['default_color'])
            gasstationen_layer.dataProvider().addFeatures([new_feature])
```

 # Hier wird der Layer-Extent aktualisiert, was notwendig ist wenn neue Features hinzugefügt werden dadurch wird sichergestellt, dass die Extents alle vorhandenen Features korrekt umfassen. „addMapLayer“ fügt dann den gasstationen_Layer zum aktuellen QGIS-Projekt hinzu, damit er in der Layerliste angezeigt wird

```
gasstationen_layer.updateExtents()
QgsProject.instance().addMapLayer(gasstationen_layer)
```

 # Schritt 2: Anwenden von Stilregeln auf den Layer "Gasnetz_AT" Hinzufügen des Attributs "line_color", falls nicht vorhanden

```
if 'line_color' not in [field.name() for field in gasnetz_layer.fields()]:
    gasnetz_layer.dataProvider().addAttributes([QgsField('line_color', QVariant.String)])
    gasnetz_layer.updateFields()

# Aktualisieren des Attributs "line_color" basierend auf "biometh_pot_percent"
with edit(gasnetz_layer):
    for feature in gasnetz_layer.getFeatures():
        biometh_pot_percent = feature['biometh_pot_percent']
        if biometh_pot_percent > 45:
            feature['line_color'] = 'grün'
            gasnetz_layer.updateFeature(feature)
```

 # Erstellen einer regelbasierten Symbologie
Wenn line_color Attribut auf „grün“ gesetzt ist = grüne Symbolsierung, für andere Features schwarze Symbolosierung

```
green_symbol = QgsLineSymbol.createSimple({'color': 'green'})
```

 # Regel für grüne Linien

```
rule1 = QgsRuleBasedRenderer.Rule(green_symbol)
rule1.filterExpression = '"line_color" = \'grün\''
rule1.label = 'Grüne Linien'
```

 # Symbol für andere Linien

```
default_symbol = QgsLineSymbol.createSimple({'color': 'black'})
```

 # Standardregel

```
default_rule = QgsRuleBasedRenderer.Rule(default_symbol)
default_rule.filterExpression = ''
default_rule.label = 'Andere Linien'
```

 # Hauptregel

```
root_rule = QgsRuleBasedRenderer.Rule(None)
root_rule.appendChild(rule1)
root_rule.appendChild(default_rule)
```

#Durch triggerRepaint() wird der Layer im QGIS Projekt aktualisiert, sodass die neue Symbologie sichtbar wird

```
renderer = QgsRuleBasedRenderer(root_rule)
gasnetz_layer.setRenderer(renderer)
gasnetz_layer.triggerRepaint()
```

 # Schritt 3: Erstellung des Layers "Biomethan_Einspeisestationen"

```
# Erstellung eines neuen Polygon-Layers
einspeisestationen_layer = QgsVectorLayer('Polygon?crs=EPSG:32633',
    'Biomethan_Einspeisestationen', 'memory')
```

```
# Hinzufügen des Attributs "color"

einspeisestationen_layer.dataProvider().addAttributes([QgsField('color', QVariant.String)])
einspeisestationen_layer.updateFields()



# Abrufen der grünen Features aus "Gasnetz_AT"

request = QgsFeatureRequest().setFilterExpression('"line_color" = \'grün\'')
green_features = gasnetz_layer.getFeatures(request)



# Iteration über die grünen Features und Erstellung von Quadraten an den Endpunkten

for feature in green_features:
    geom = feature.geometry()
    if geom.isMultipart():
        lines = geom.asMultiPolyline()
        for line in lines:
            start_point = line[0]
            end_point = line[-1]
            for point in [start_point, end_point]:
                square_geom = create_square_at_point(point, 50)
                new_feature = QgsFeature()
                new_feature.setGeometry(square_geom)
                new_feature.setAttributes(['default_color'])
                einspeisestationen_layer.dataProvider().addFeatures([new_feature])
    else:
        line = geom.asPolyline()
        start_point = line[0]
        end_point = line[-1]
        for point in [start_point, end_point]:
            square_geom = create_square_at_point(point, 50)
            new_feature = QgsFeature()
            new_feature.setGeometry(square_geom)
            new_feature.setAttributes(['default_color'])
            einspeisestationen_layer.dataProvider().addFeatures([new_feature])

# Aktualisieren der Layer-Extents und Hinzufügen zum Projekt
einspeisestationen_layer.updateExtents()
QgsProject.instance().addMapLayer(einspeisestationen_layer)
```

3.5.2 Durchführung Experiment Testszenario 2

3.5.2.1 Prompt für Testszenario 2:

Erstelle einen Python Code für die QGIS Python Konsole:

Im aktuellen QGIS-Projekt gibt es einen Layer „gebaeude_kloburg" mit Polygon-Features, die Gebäude darstellen.

1. Erstelle einen neuen Layer „Versorgungsnetzwerk" im Koordinatensystem: EPSG: 4326, mit Line-Features, der Stromleitungen zwischen den Gebäuden repräsentiert.
- Diese Leitungen sollen die nächstgelegenen Gebäude miteinander verbinden und dürfen nicht länger als 500 Meter sein.
2. Wenn ein Gebäude mehr als 3 Verbindungen hat, erstelle einen neuen Layer „Umspannstationen" im Koordinatensystem: EPSG: 4326, der an diesem Gebäude ein Symbol (Punkt-Feature) darstellt.

Abbildung 17: Prompt Testszenario 2 in ChatGPT (Eigene Darstellung 2024)

Der in Abbildung 17 dargestellte Prompt wurde im folgenden Schritt jeweils 20-mal für jedes einzelne Modell übergeben in den Abbildungen 19, 20 findet sich eine korrekte Lösung des Modells GPT o1 Preview. In Abbildung 18 sind die Ausgangsdaten in Form von Open Street Map Gebäude als Polygon-Features in Orange eingefärbt zu sehen.

Abbildung 18: Screenshot in QGIS mit Gebäude-Polygonen in Klosterneuburg und OSM-Hintergrundkarte (Eigene Darstellung 2024)

Nach Ausführung des Codes sollte sich nun wie in Abbildung 19 zu sehen ein Layer mit Linefeatures, hier in rot gefärbt im QGIS-Projekt befinden. Dieser Layer verbindet die Wohngebäude mit Stromleitungen.

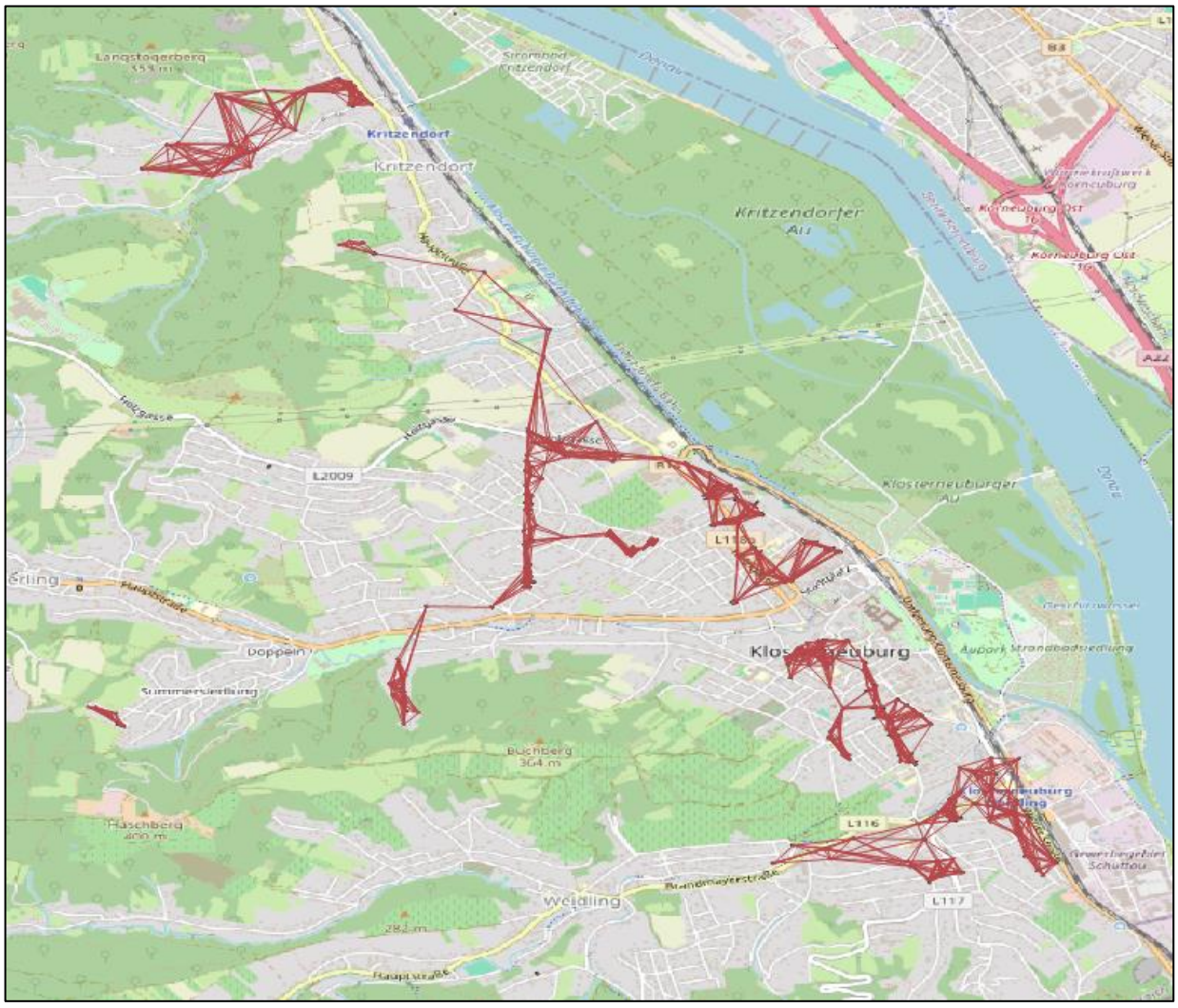

Abbildung 19: Screenshot in QGIS mit Stromleitungs-Features (rot gefärbt) und Hintergrundkarte (Eigene Darstellung 2024)

In Abbildung 20 ist zu sehen, dass ein weiterer Layer mit Punkt-Features angelegt wurde, der Umspannstationen an Gebäuden symbolisiert, die mehr als 3 Verbindungen aufweisen.

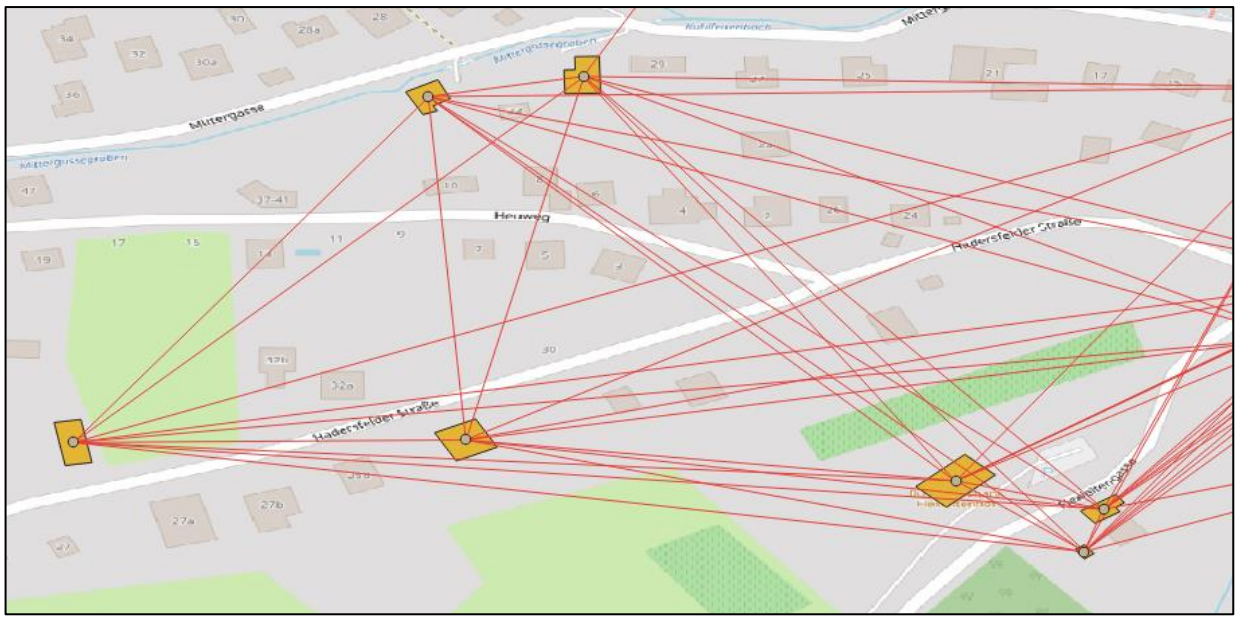

Abbildung 20: Screenshot in QGIS mit Umspannstation-Polygonfeatures und Leitungsfeatures (Eigene Darstellung 2024)

#Zunächst müssen die notwendigen Module importiert werden hervorzuheben ist hier „QgsSpatialIndex“ Was verwendet wird, um einen räumlichen Index zu erstellen, der es ermöglich schnell die nächstgelegenen Gebäude (innerhalb von 500 Metern zu finden) ohne wäre diese Operation sehr rechenintensiv. Das Modul QgsDistanceArea, sorgt für die korrekte Streckenberechnung der Stromleitungen

```
from qgis.core import (
    QgsProject,
    QgsFeature,
    QgsGeometry,
    QgsPoint,
    QgsPointXY,
    QgsSpatialIndex,
    QgsCoordinateReferenceSystem,
    QgsCoordinateTransform,
    QgsField,
    QgsFields,
    QgsVectorLayer,
    QgsWkbTypes,
    QgsDistanceArea
)
from PyQt5.QtCore import QVariant
```

```
# 1. Zugriff auf den im  Projekt geladenen Layer "gebaeude_kloburg"

gebaeude_layer = QgsProject.instance().mapLayersByName('gebaeude_kloburg')[0]


# 2. Zugriff auf Quell- und Ziel-Koordinatensystem (EPSG:4326)

source_crs = gebaeude_layer.crs()
target_crs = QgsCoordinateReferenceSystem('EPSG:4326')


# 3. Koordinatentransformator für den korrekten Umgang mit Koordinatensystemen

coord_transform = QgsCoordinateTransform(source_crs, target_crs, QgsProject.instance())


# 4. Bereiten Sie die Sammlung von Zentroid-Features und den räumlichen Index vor

centroid_features = []
spatial_index = QgsSpatialIndex()
feature_centroids = {}  # Schlüssel: Feature-ID, Wert: Zentroid-Punkt
centroid_feature_dict = {}  # Schlüssel: fid, Wert: Zentroid-Feature


# 5. Erstellung der Zentroiden, transformieren zu EPSG:4326, Aufbau des räumlichen Index

for feature in gebaeude_layer.getFeatures():
    geom = feature.geometry()
    centroid = geom.centroid()




# Transformieren der Zentroiden zum EPSG:4326

    centroid.transform(coord_transform)


# Erstellen Sie ein neues Feature mit dem Zentroid-Punkt

    centroid_feature = QgsFeature()
    centroid_feature.setGeometry(centroid)
    centroid_feature.setAttributes([feature.id()])
    centroid_feature.setId(feature.id())
```

 # Setzen der ID auf die ursprüngliche Feature-ID

```
    centroid_features.append(centroid_feature)
    centroid_feature_dict[feature.id()] = centroid_feature
    # Zum räumlichen Index hinzufügen
    spatial_index.addFeature(centroid_feature)
    # Zentroid-Geometrie speichern
    feature_centroids[feature.id()] = centroid_feature.geometry().asPoint()
```

 # 6. Vorbereitung des Distanzberechners, hier wird festgelegt, dass für Distanzberechnungen das WGS84 Ellipsoid verwendet wird, damit Berechnungen nicht in Grad, sondern in realen Entfernungen ausgeführt werden, die die Krümmung der Erde berücksichtigen

```
distance_calculator = QgsDistanceArea()
distance_calculator.setSourceCrs(target_crs, QgsProject.instance().transformContext())
distance_calculator.setEllipsoid('WGS84')
```

 # 7. Finden von Verbindungen zwischen Gebäuden innerhalb von 500 Metern

```
connections = {}  # Schlüssel: Tupel von sortierten fids, Wert: Distanz
connection_counts = {}  # Schlüssel: fid, Wert: Anzahl der Verbindungen

for feature in centroid_features:
    fid = feature.id()
    point = feature.geometry().asPoint()
    # Nahegelegene Features mithilfe des räumlichen Index abrufen
    nearby_ids = spatial_index.nearestNeighbor(point, 10)  # Bis zu 10 nächste
    for near_fid in nearby_ids:
        if near_fid == fid:
            continue  # Überspringe sich selbst
        # Nahegelegenes Zentroid-Feature abrufen
        near_feature = centroid_feature_dict[near_fid]
        near_point = near_feature.geometry().asPoint()
        # Distanz berechnen
        dist = distance_calculator.measureLine(point, near_point)
        if dist <= 500:
            # Überprüfen, ob die Verbindung bereits existiert
            connection = tuple(sorted([fid, near_fid]))
            if connection not in connections:
                connections[connection] = dist
                # Aktualisieren der Verbindungsanzahl
                connection_counts[fid] = connection_counts.get(fid, 0) + 1
                connection_counts[near_fid] = connection_counts.get(near_fid, 0) + 1
        else:
```

```
        continue


# 8. Erstellung des Line-Features-Layer "Versorgungsnetzwerk" in EPSG:4326

line_layer = QgsVectorLayer("LineString?crs=EPSG:4326", "Versorgungsnetzwerk", "memory")
line_provider = line_layer.dataProvider()


# Speichern der ID des ersten verbundenen Features „fid1“ und des zweiten „fid2“ als Integer und der Distanz zwischen den Features „distance_m“ als double und hinzufügen zum „Versorgungsnetzwerk“ Layer

fields = QgsFields()
fields.append(QgsField("fid1", QVariant.Int))
fields.append(QgsField("fid2", QVariant.Int))
fields.append(QgsField("distance_m", QVariant.Double))
line_provider.addAttributes(fields)
line_layer.updateFields()



# Linien-Features werden zwischen den Zentroiden der Gebäude erstellt und die Attribute fid1, fid2 und distance_m gesetzt

line_features = []

for connection, distance in connections.items():
    fid1, fid2 = connection
    point1 = feature_centroids[fid1]
    point2 = feature_centroids[fid2]
    line = QgsGeometry.fromPolylineXY([QgsPointXY(point1), QgsPointXY(point2)])
    line_feature = QgsFeature()
    line_feature.setGeometry(line)
    # Attribute setzen
    line_feature.setAttributes([fid1, fid2, distance])
    line_features.append(line_feature)

# Features werden zum Linien-Layer hinzugefügt
line_provider.addFeatures(line_features)
line_layer.updateExtents()

# Linien-Layer werden zum Projekt hinzugefügt

QgsProject.instance().addMapLayer(line_layer)


# 9. Gebäude mit mehr als 3 Verbindungen werden identifiziert
```

```
umspann_fids = [fid for fid, count in connection_counts.items() if count > 3]
```

 # 10. Punkt-Layer "Umspannstationen" in EPSG:4326 wird erstellt

```
umspann_layer = QgsVectorLayer("Point?crs=EPSG:4326", "Umspannstationen", "memory")
umspann_provider = umspann_layer.dataProvider()
```

 # Fügt dem Layer „Umspannstationen“ ein neues Attributfeld "fid" hinzu, dass die ID des zugehörigen Gebäudes speichert. Dadurch wird die Zuordnung zwischen Umspannstationen und Gebäuden ermöglicht.

```
fields = QgsFields()
fields.append(QgsField("fid", QVariant.Int))
umspann_provider.addAttributes(fields)
umspann_layer.updateFields()
```

 # Punkt-Features für „Umspann“ Layer erstellen

```
umspann_features = []

for fid in umspann_fids:
   point = feature_centroids[fid]
   point_feature = QgsFeature()
   point_feature.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(point)))
   point_feature.setAttributes([fid])
   umspann_features.append(point_feature)
```

 # Features zum Umspann-Layer hinzugefügt und anschließend Layer zum Projekt hinzugefügt

```
umspann_provider.addFeatures(umspann_features)
umspann_layer.updateExtents()


QgsProject.instance().addMapLayer(umspann_layer)
```

3.5.3 Durchführung Experiment Testszenario 3

3.5.3.1 Prompt für Testszenario 3

Aufgabenstellung:

Erstelle ein Python Skript für die QGIS Python Konsole:

Im aktuellen QGIS-Projekt gibt es einen Layer „H2_Leitungen_Houston" mit Line-Features, die Wasserstoffleitungen in Österreich darstellen.
Erstelle einen neuen Layer „Hydrolyse_Stationen", der Punkt-Features enthält und die Stationen repräsentiert, an denen Wasserstoff produziert wird. Diese Stationen sollen an den Schnittpunkten der Wasserstoffleitungen liegen (Werkzeug: Intersect).
Verwende das Buffer-Werkzeug, um um jede Wasserstoffleitung einen 100 Meter breiten Puffer zu erstellen. Der Puffer soll als neuer Layer „H2_Puffer" gespeichert werden.
Stelle sicher, dass das Koordinatensystem des neuen Layers EPSG:32140 ist.

Abbildung 21: Prompt für Testszenario 3 in ChatGPT (Eigene Darstellung 2024)

Der Prompt aus Abbildung 21 wurde anschließend wieder den Modellen übergeben, folgend wird eine korrekte Lösung des Modells Deepseek Coder 2.5 visualisiert. In Abbildung 22 sind zunächst die unbearbeiteten Ausgangs-Features, also Wasserstoffleitungen in Houston (USA) lokalisiert, zu sehen.

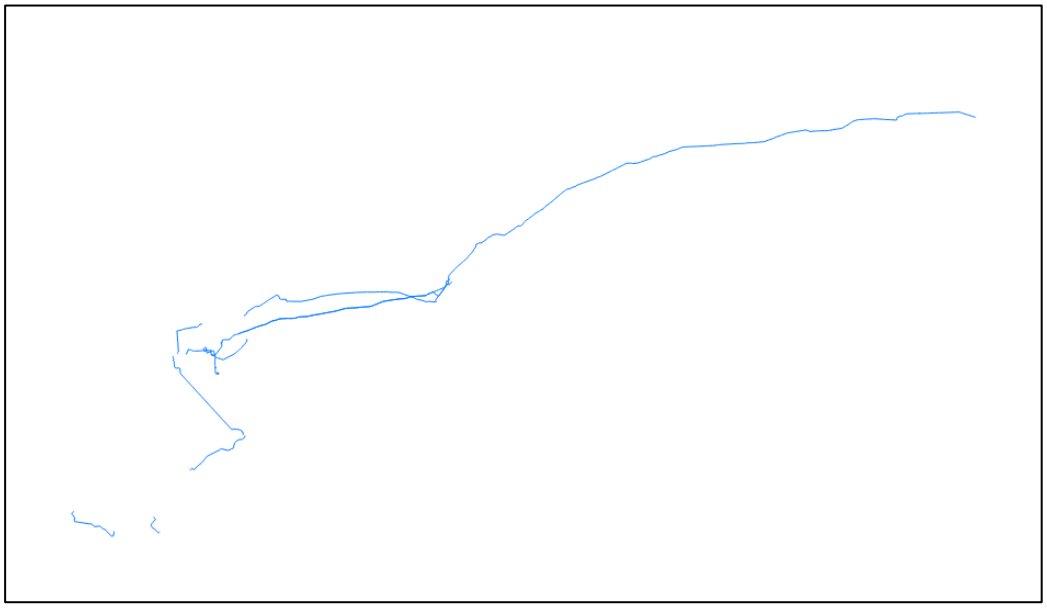

Abbildung 22: Screenshot in QGIS von Wasserstoff-Leitungsfeatures in Houston (Eigene Darstellung 2024)

In Abbildung 23 sind die erstellten Hydrolyse-Stationen an den Schnittpunkten der Wasserstoffleitungen visualisiert

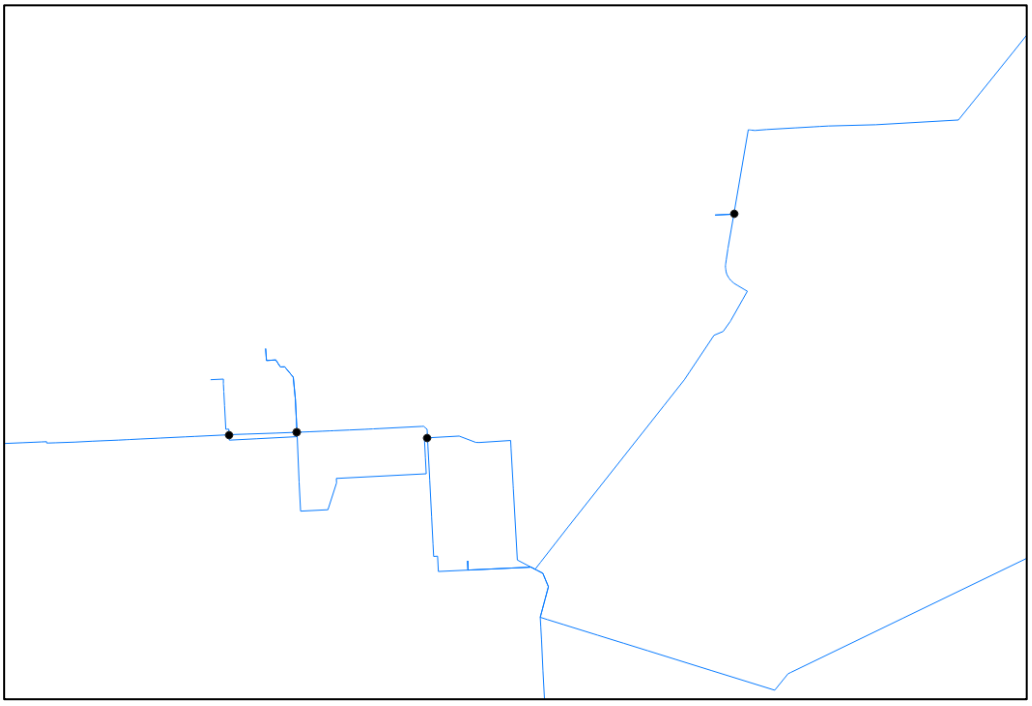

Abbildung 23: Screenshot in QGIS von Hydrolyse Stationen an Schnittpunkten (Eigene Darstellung 2024)

In Abbildung 24 wird nun noch der Puffer um die Leitungen in Orange dargestellt.

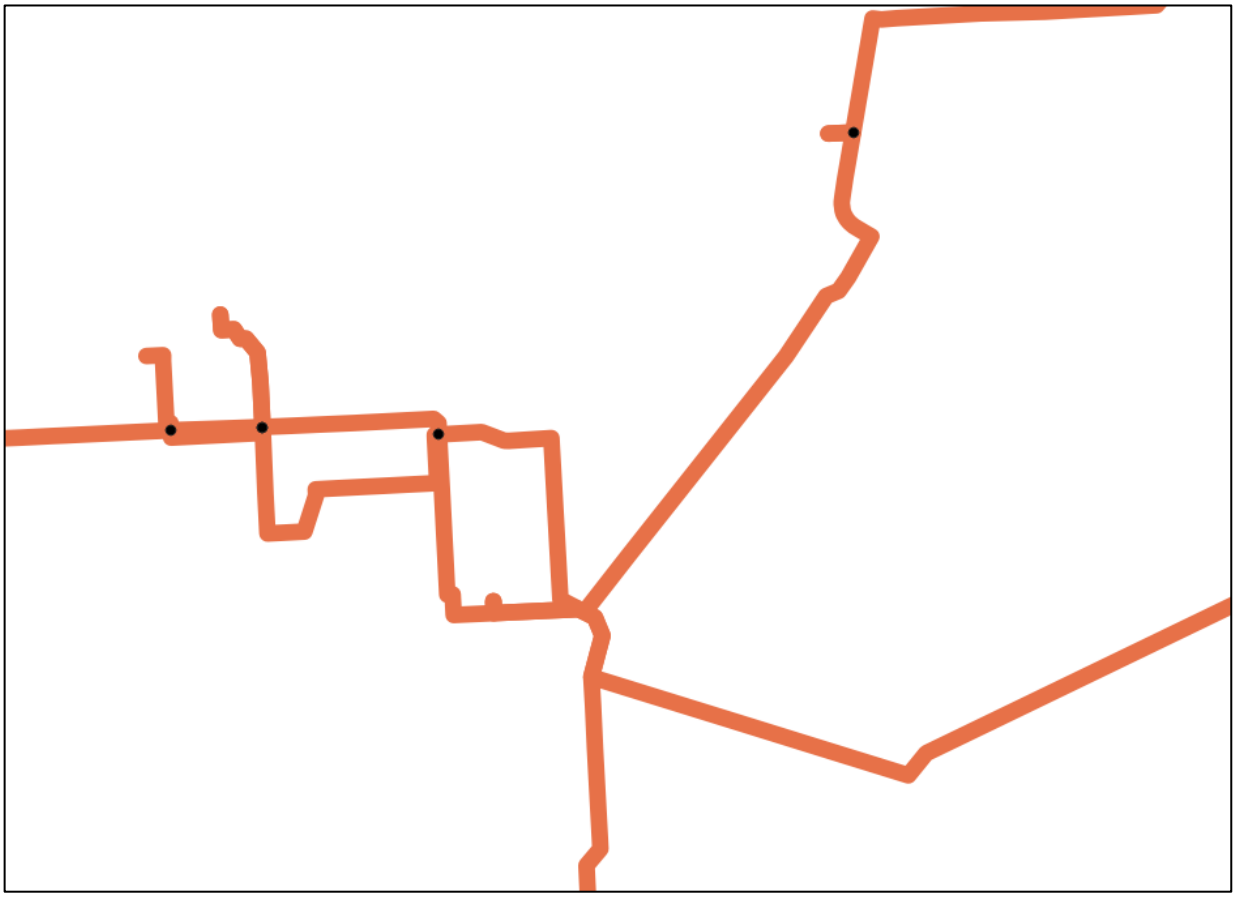

Abbildung 24: Screenshot in QGIS von Orange eingefärbten Pufferfeatures um Leitungsfeatures (Eigene Darstellung 2024)

```
# Testszenario 3 QGIS Python Konsole Skript


# Layer "H2_Leitungen_Houston" wird geladen
h2_leitungen_layer = QgsProject.instance().mapLayersByName('H2_Leitungen_Houston')[0]


# Schnittpunkte der Wasserstoffleitungen werden ermittelt
intersect_points = processing.run("native:lineintersections", {
   'INPUT': h2_leitungen_layer,
   'INTERSECT': h2_leitungen_layer,
   'OUTPUT': 'memory:'
})['OUTPUT']


# Neuen Layer "Hydrolyse_Stationen" erstellen
hydrolyse_stationen_layer = QgsVectorLayer("Point?crs=EPSG:32140", "Hydrolyse_Stationen", "memory")
QgsProject.instance().addMapLayer(hydrolyse_stationen_layer)


# Schnittpunkte werden  in den neu erstellten Layer kopiert
features = []
for feature in intersect_points.getFeatures():
   geom = feature.geometry()
   new_feature = QgsFeature()
   new_feature.setGeometry(geom)
   features.append(new_feature)


hydrolyse_stationen_layer.dataProvider().addFeatures(features)


#  100 Meter Puffer um die Wasserstoffleitungen wird erstellt
buffered_layer = processing.run("native:buffer", {
   'INPUT': h2_leitungen_layer,
```

```
    'DISTANCE': 100,
    'SEGMENTS': 5,
    'END_CAP_STYLE': 0,  # FLAT
    'JOIN_STYLE': 0,    # MITER
    'MITER_LIMIT': 2,
    'DISSOLVE': False,
    'OUTPUT': 'memory:'
})['OUTPUT']


# Der neue Layer "H2_Puffer" wird erstellt und die Puffer Features anschließend in den Layer kopiert
h2_puffer_layer = QgsVectorLayer("Polygon?crs=EPSG:32140", "H2_Puffer", "memory")
QgsProject.instance().addMapLayer(h2_puffer_layer)
features = []
for feature in buffered_layer.getFeatures():
    geom = feature.geometry()
    new_feature = QgsFeature()
    new_feature.setGeometry(geom)
    features.append(new_feature)


h2_puffer_layer.dataProvider().addFeatures(features)
print(f"Koordinatensystem des Layers 'H2_Puffer': {h2_puffer_layer.crs().authid()}")
```

4 Ergebnisse und Diskussion

Im folgenden Kapitel werden die Ergebnisse der Experimente diskutiert.

4.1 Präsentation der Ergebnisse

Die folgende Tabelle 4 zeigt, wie die drei untersuchten KI-Sprachmodelle in verschiedenen Tests abschneiden. Die Tests messen Pass@k und Accuracy über drei Testfälle T1, T2 und T3. Zudem wird ein durchschnittlicher Pass@k-Wert aufgeführt. Der Pass@k-Wert gibt dabei einen Wert an, der bedeutet, dass man mit einer Wahrscheinlichkeit von beispielsweise 70 **%** mindestens eine korrekte Lösung bekommt, wenn man zufällig **k** Lösungen aus den insgesamt **n** generierten Lösungen auswählt.

GPT o1 preview erzielte die höchste Gesamtleistung mit einem Pass@k-Durchschnitt von 0.628. Es erreicht in allen Tests höhere Werte gegenüber den anderen Modellen. Besonders in Testszenario 1 und 3 mit Pass@k von 0.7183 und einer Accuracy von 0.8. DeepSeek Coder erreicht einen Pass@k-Durchschnitt von 0.2325 und liegt damit weit hinter GPT o1 preview. Auffällig ist, dass es in T2 einen Wert von 0.0 aufweist. Bei T3 schneidet es moderat ab: Pass@k beträgt 0.4474, Accuracy liegt bei 0.6. Claude 2.5 Sonett zeigt von allen Modellen die schlechteste Leistung. Der Pass@k-Durchschnitt beträgt nur 0.2003. Bei T1 und T2 erreicht es keine erfolgreichen Tests. Auch in T3 bleibt die Erfolgsrate niedrig, mit 0.1234.

Model	T1 Pass@k	T1 AC	T2 Pass@k	T2 AC	T3 Pass@k	T3 AC	Ø Pass@k
GPT o1p	0.7183	0.8	0.4474	0.4	0.7183	0.8	0.628
DeepSeek Coder	0.25	0.2	0.0	0.0	0.4474	0.6	0.2325
Claude 2.5 Sonett	0.0	0.0	0.0	0.0	0.1234	0.6	0.2003

Tabelle 4: Testergebnisse mit Pass@k und Accuracy (Eigene Darstellung)

4.2 Fehleranalyse

4.2.1 GPT o1 preview Fehler

4.2.1.1 Aufgetretene Fehler im Testszenario 1

Prompt 2:

Attribute Error: Verwendung der Methode topLeft, die in der QGIS Python API Klasse QGSRectangle nicht vorhanden ist (Class: QgsRectangle, 2025j). Die Methode topLeft stammt aus der Klasse QRect Class vom plattformübergreifenden Framework „QT“ für grafische Benutzeroberflächen (QRect Class | Qt Core 6.8.1, 2025n). Zwar kann man im Kontext von QGIS durchaus auf diese Methoden zugreifen, insbesondere um grafische Nutzeroberflächen für Erweiterungen der QGIS-Funktionalitäten zu programmieren, doch sollten, sobald es um geometrische Operationen geht, die QGIS Python API Methoden verwendet werden, da diese darauf spezialisiert sind.

4.2.1.2 Aufgetretene Fehler im Testszenario 2

Prompt 1:

Attribute Error: Im Code wird die Methode clone() verwendet, um Features zu kopieren, um die nachfolgenden Transformationen nicht auf den Originaldaten ausführen zu müssen. Diese Methode ist nicht Teil der QgsGeometry-Klasse, sondern wird in anderen Klassen wie Qgs VectorLayer zum Kopieren von Features verwendet. In QgsGeometry wird statt clone() die Methode copy() verwendet. (Class: QgsGeometry, 2025f). Durch diesen Fehler kommt es zum Abbruch des Codes und der Layer „Umspannstationen“ wird gar nicht erst erstellt.

Prompt 14:

Attribute Error: Hier wird irrtümlich die Methode setEllipsoidalMode() verwendet, welche in QGIS 3.x nicht mehr Teil der QgsDistanceArea-Klasse ist (Class: QgsDistanceArea, 2025d). In früheren QGIS-Versionen wurde diese Methode noch verwendet und sollte in diesem Code

sicherstellen, dass die Erdkrümmung bei Messungen wie sie notwendig sind, um die Maximallänge von 500 Metern der Stromleitungen zu gewährleisten, berücksichtigt wird. Der Fehler bringt das Skript zum Abbruch und es werden keine Layer erstellt.

Prompt 18:

Type Error: Im Code wird versucht ein Spatial Index zu erstellen, um zeitsparend Nachbarschaftsbeziehungen der Wohngebäude zu ermitteln. Hierbei wird direkt an die Klasse QgsSpatialIndex() eine Liste mit Punkten zu übergeben, was aber laut Python API Dokumentation nicht erlaubt ist, richtig wäre zuerst QgsSpatialIndex zu erstellen und dann mit der Methode .addFeatures Features zu übergeben (Class: QgsSpatialIndex, 2025k). Auch hier wird das Skript durch den Fehler gestoppt und es kommt zu keiner Layer-Erstellung.

4.2.1.3 Aufgetretene Fehler im Testszenario 3

Prompt 20:

Logical Error: Im Code wird QgsProcessingFeatureSourceDefinition verwendet, um den in QGIS geladenen Ausgangslayer mit den Wasserstoffleitungen als Wrapper zu laden. Diese Vorgehensweise ermöglicht es beim Laden des Layers mehrere Optionen zu haben, beispielsweise nur einen Teil der Features zu laden. Die Default-Einstellung führt aber dazu das keine Features geladen werden, woraufhin auch in den erstellten Layern mit den Hydrolyse-Stationen und Puffern keine Features resultieren. Die richtige Vorgehensweise, entweder den Layer direkt ohne Wrapper zu laden oder den Wrapper auf „False“ zu setzen, damit alle Features ordnungsgemäß geladen und übertragen werden (Class: QgsProcessingFeatureSourceDefinition, 2025h).

4.2.2 Deepseek Coder Fehler

4.2.2.1 Aufgetretene Fehler im Testszenario 1

Prompt 1, 2, 11, 19:

Attribute Error: Hier wird irrtümlich die Methode vertexCount() verwendet, um an die Start- und Endpunkte der Linien zu gelangen, um dort Polygone zu zeichnen. Die Methode ist in QGIS

3.x nicht in der Klasse QGSGeometry vorhanden, stattdessen verwendet man beispielsweise asMultiPolyline() um die Stützpunkte von mehreren Linien zu erhalten (Class: QgsGeometry, 2025f)

4.2.2.2 Aufgetretene Fehler im Testszenario 2

Prompt 1:

Type Error: Im Skript wird versucht, aus Gebäudezentroid-Punkten ein Linestring zu generieren, der die unter 500 Meter Stromlinien symbolisiert. Da aber mit QgsGeometry.asPoint() ein QgsPointXY erzeugt wird und später mit QgsGeometry.asPolyline versucht wird, daraus ein Linefeature zu generieren, bricht das Skript ab, da eigentlich QgsPoint statt QgsPointXY erwartet wird und hätte somit zunächst in diesen Typ umgewandelt werden müssen.

Prompt 6:

Logical Error: Zwar ist das Skript ausführbar und die Layer Versorgungnetzwerk und Umspannstationen werden erstellt und mit Features befüllt. Dennoch wird die Angabe, dass die Leitungen nicht länger als 500 Meter sein dürfen, nicht ordnungsgemäß umgesetzt. Das liegt daran, dass im geforderten Koordinatenreferenzsystem EPSG: 4326 in Grad gemessen wird. Um dennoch in Metern messen zu können, hätte man beispielsweise die Klasse QgsDistanceArea mit der Methode measure.line() verwenden können, um ein Bezugsellipsoid (WGS84) zu setzen (Class: QgsDistanceArea, 2025e). Damit wäre die Distanz korrekt berechnet.

Prompt 7:

Type Error: Im Skript wird durch die Methode .getfeatures() der QgsFeatureIterator verwendet, um die Features im Umspannstationen-Layer nach und nach und damit speicheroptimiert zu laden, statt alle auf einmal. Im Code wird versucht len() darauf anzuwenden, was nicht erlaubt ist, da der FeatureIterator die Feature on demand lädt (Class: QgsVectorLayer, 2025l). Eine gültige Vorgehensweise wäre gewesen, den FeatureIterator in eine Liste umzuwandeln, zum Beispiel so: list(umspannstationen_layer.getFeatures().

Prompt 10:

Type Error: Hier wird versucht, den Gebäudepolygonlayer mit der asPoint() Methode als Punktgeometrie zu verwenden, um die Distanzen der Gebäude zu berechnen. Die ist keine gültige Vorgehensweise, da Polygone nicht direkt als Punkte verwendet werden können (Class: QgsGeometry, 2025f), es müssten zunächst die Zentroide berechnet werden, z.B. mit centroid().

Prompt 13:

Logical Error (siehe Prompt 6)

4.2.2.3 Aufgetretene Fehler im Testszenario 3

Prompt 4, 6, 10:

Type Error:

Hier wird irrtümlich der Algorithmus native:intersection statt native:lineintersection. Der Output davon ist vom gleichen Typ wie der Input, also entstehen Linefeatures, die dann in weiterer Folge mit der Methode .asPoint() vergeblich als Punkte weiterverarbeitet werden. Es resultiert ein Abbruch des Skripts (Vector overlay — QGIS Documentation documentation, 2025a)

4.2.3 Claude Sonnet 3.5 Fehler

4.2.3.1 Aufgetretene Fehler im Testszenario 1

Prompt 1, 16, 20:

Type Error: Der Code behandelt die Multilinestrings des Gasnetz-Layers als Linestring und will die Methode asPolyline() anwenden, dies führt zu einem Abbruch. Stattdessen wäre die Methode as MultiPolyline() für eine korrekte Verarbeitung notwendig gewesen (*Class: QgsGeometry,* 2025f)

***Prompt 7**:*

Assertion Error: Hierbei entsteht ein Assertion Error, weil die qgis.core Klasse „edit()“ mit with aufgerufen wird, dadurch wird intern eine assert-Überprüfung durchgeführt, problematisch ist dies, wenn der Layer nicht in den Editiermodus gesetzt werden kann, weil es sich beispielsweise um einen Datenbanklayer handelt, für den keine Schreibrechte vorhanden sind oder wie in unserem Fall der Layer sich schon im Editiermodus befand. Dies führte zum Abbruch des Skripts (*Class: edit,* 2025c).

Prompt 10:

Attribute Error: Fehlerhafte Methode vertexCount (siehe Deepseek Coder T1)

4.2.3.2 Aufgetretene Fehler im Testszenario 2

Prompt 2:

Attribute Error: Der Code versucht am Ende der Operationen die Methode .mapCanvas() auf QgsProject.instance() anzuwenden, um den Karteninhalt zu aktualisieren, damit die neuen Daten sichtbar sind. Diese Methode gehört allerdings nicht zur Klasse QgsProject (Class: QgsProject, 2025i). Dieser Fehler bringt das Skript aber nicht zum Absturz und die Layer werden ordnungsgemäß erstellt, dennoch ist die Länge und Anzahl der Leitungen nicht korrekt, weil der gleiche Logical Error besteht wie bei DeepSeek Coder T2 Prompt 6.

Prompt 5:

Type Error (siehe Deepseek Coder T2 Prompt 10)

Prompt 9:

Logical Error (siehe Deepseek Coder T2 Prompt 6)

Prompt 15:

Logical Error (siehe Deepseek Coder T2 Prompt 6)

Prompt 20:

Attribute Error: Der Code versucht, die Methode .nearestNeighbor() aufzurufen, um die Nachbarschaftsbeziehungen der Gebäude zu ermitteln. Die Klasse QgsVectorLayer hat aber keine integrierte Methode, um nächstgelegene Features zu finden (*Class: QgsVectorLayer,* 2025l).

4.2.3.3 Aufgetretene Fehler im Testszenario 3

Prompt 9:

Logical Error: Die falsche Schreibwese des Algorithmus „native:linesintersections“ (statt korrekt native:lineintersection) bringt das Skript zum Absturz (*28.1.23. Vector overlay — QGIS Documentation,* 2025a)

Prompt 17:

Logical Error: Hier läuft das Skript eigentlich korrekt, nur wurden die Ausgangslayer nicht korrekt in H2_Hydrolyse_Stationen und „H2_Puffer“ benannt. Die Namen der temporären Memory-Layer passen sich nicht automatisch den Ausgabepfaden oder Verarbeitungsinstanzen an. Das Skript speichert die Shapefiles zwar als "Hydrolyse_Stationen.shp" und "H2_Puffer.shp". Der Memory-Layer im QGIS-Projekt behält aber seinen ursprünglichen Namen. Er verwendet entweder den Standard oder einen generischen Namen des Algorithmus. Mit der Methode .setName ist stattdessen eine korrekte Benennung möglich (*Class: QgsMapLayer,* 2025g).

4.2.4 Kategorisierung der Fehler

In der folgenden Tabelle 5 werden die identifizierten Fehler nach ihrer Fehlerkategorie über die Modelle hinweg aufgelistet.

Modell	T1	T2	T3
GPT o1p	Attribute Error → Prompt: 2	Attribute Error → Prompt: 1, 14	Attribute Error → none
	Type Error → none	Type Error → Prompt: 18	Type Error → Prompt: 20
	Logical Error → none	Logical Error → none	Logical Error → Prompt: 20
	Assertion Error → none	Assertion Error → none	Assertion Error → none
Deepseek Coder	Attribute Error → Prompt: 1, 2, 11, 19	Attribute Error → none	Attribute Error → none
	Type Error → none	Type Error → Prompt: 1, 7	Type Error → Prompt: 4, 6, 10
	Logical Error → none	Logical Error → Prompt: 6	Logical Error → Prompt: none
	Assertion Error → none	Assertion Error → none	Assertion Error → none, 15
Claude Sonnet 3.5	Attribute Error → Prompt: 10	Attribute Error → Prompt: 2, 20	Attribute Error → none
	Type Error → Prompt: 1, 16, 20	Type Error → Prompt: 5	Type Error → none
	Logical Error → none	Logical Error → Prompt: 9, 15	Logical Error → Prompt: 9, 17
	Assertion Error → Prompt: 7	Assertion Error → none	Assertion Error → none

Tabelle 5: Übersichtliche Einteilung nach Fehlerkategorie (Eigene Darstellung)

4.3 Interpretation und Diskussion

Wie die Fehleranalyse und Kategorisierung der Fehler nahelegen, sind Fehler der Modelle auf unterschiedlichen Ebenen wie Syntax, konzeptueller Planung des Workflows oder auch der korrekten Python API-Nutzung aufgetreten. Der **TypeError** ist die Fehlerkategorie, die am häufigsten über alle Modelle hinweg aufgetreten ist **(11 Mal)**. Beispielsweise die irrtümliche Verwendung von Multipolylines als Polyline. Dies deutet darauf hin, dass die Modelle zwar häufig formal korrekten Code generieren können, aber nicht auf Geometrie-Typen optimiert sind. Des Weiteren fällt auf, dass häufig **Attribute Error (10 Mal)** entstehen, weil der generierte Code Methoden verwendet, die nicht mehr existieren oder nicht Teil der QGIS Python API sind und somit nicht für geometrische Operationen optimiert sind. Dies könnte daraus entstehen, dass die verwendeten LLM‘s auf große Datensätze mit unspezifischen Code-Repositories trainiert sind, darunter beispielsweise Foreneinträge in denen möglicherweise Methoden verwendet werden, die mit älteren QGIS-Versionen kompatibel sind, aber in QGIS 3.x nicht mehr funktional sind. Der dritthäufigste Fehler ist der **Logical Error (6) mal**. Hier fällt auf, dass dieser Fehler vor allem im Testszenario 2 bei den Modellen Deepseek Coder und Claude Sonett 3.5 aufgetreten ist. Im Testszenario 2 wurde im Prompt bewusst zur Erhöhung der Komplexität das EPSG: 4326 Koordinatensystem gefordert, was bekanntlich in Grad misst. Da diese Modelle dies bei ihrer Distanzberechnung nicht berücksichtigen, kommen fehlerhafte Ergebnisse heraus. Daraus lässt sich schließen, dass sich im Trainingsmaterial der Modelle nur wenige oder keine Beispiele befinden, die diese Thematik berücksichtigen. Zwar befindet sich der zu berücksichtigende Kontext im Prompt, dennoch scheinen die Modelle die Abfolge des Codes zu verfolgen, den sie am häufigsten aus ihren Trainingsbeispielen kennen. Das einzige der drei Modelle, das diesen Kontext „verstanden“ hat ist GPT o1 preview was wie in der Beschreibung der Modelle aufgeführt die Tree of Thought und Chain of Thought-Techniken verwendet, was die gleichzeitige Prozessierung von mehreren Lösungspfaden und das zurückspringen zu vorherigen Schritten ermöglicht, es scheint, als ob sich so, Aufgaben wie im Testszenario 2, wo gegenüber den anderen Testszenarien komplexeres logisches Vorgehen erfordert ist, besser lösen lassen. Aus den Fehlern der Modelle lassen sich folgende Verbesserungsvorschläge für das Arbeiten mit LLM’s in diesem Kontext ableiten. Durch optimiertes Prompt Engineering, dergestalt, dass der Mangel an Domänenwissen der Modelle durch spezifische Hinweise auf zu verwendende Geometrietypen oder Methoden der QGIS Python API ausgeglichen wird, sollten bessere Ergebnisse sichergestellt werden. Falls beim Anwender konkretes Wissen über die zu verwendenden Methoden für die Aufgaben vorhanden ist, können die Modelle explizit im Prompt daraufhin gewiesen werden, welche Methoden für welche Aufgabe geeignet sind, was die Fehlerquote verringern sollte. Für Anwender ohne Programmierkenntnisse und API-Wissen bleibt dagegen die Möglichkeit, den beim Ausführen erhaltenen Errorcode in einem weiteren Prompt dem Modell zur Verfügung zu stellen, in manchen Fällen kann dies dazu führen, dass das Modell eine korrekte angepasste Version generiert. Die Erkenntnisse aus der Fehleranalyse

decken sich mit den Forschungsarbeiten (Gramacki et al., 2024, S. 7) die hervorheben, dass die Modelle zwar generell für die Generierung von Code geeignet sind, jedoch oft Schwierigkeiten mit Geodatenverarbeitungsspezifischem Wissen haben, weil hierbei spezifische Bibliotheken verwendet werden, die nicht im gleichen Maß Teil der Trainingsdaten von LLMS sind wie es allgemeinere z.B. Data-Science Bibliotheken sind und damit die Modelle in manchen Fällen wichtige Geodatenverarbeitungstools nicht verwenden (Gramacki et al., 2024, S. 1–7). Des Weiteren kam eine Untersuchung, die LLM's einem GIS-Examen unterzogen, zum Schluss, dass die Modelle Probleme bei numerischen Berechnungen haben, insbesondere bei der Umrechnung von Einheiten Bogenmaß in Grad (Mooney et al., 2023, S. 92. Diese Thematik konnte in der Untersuchung im Testszenario 2 dieser Arbeit ebenfalls identifiziert werden, als die Modelle irrtümlich Längenberechnungen in Grad ohne Transformation oder Ellipsoid-Berücksichtigung durchführten.

4.3.1 Limitation der Forschung

Trotz fundierter Erkenntnisse ist zu beachten, dass die in dieser Arbeit verwendete Vorgehensweise in mehreren Punkten beschränkt ist. Da in der Untersuchung aus Ressourcengründen nur eine kleine Zahl von Modellen berücksichtigt werden konnte und mittlerweile eine Vielzahl verschiedener Modelle mit unterschiedlichen Ansätzen und Trainingsdaten existieren ist unklar, ob die Ergebnisse übertragbar sind oder bei anderen LLM´s stark abweichende Ergebnisse resultieren würden. Ein weiterer Umstand ist die rasante technische Entwicklung im Forschungsfeld der Large Language Models. In dieser Arbeit konnten nur Modelle, die bis zu einem gewissen Zeitpunkt entwickelt wurden, berücksichtigt werden, im Laufe der Arbeit wurden neuere leistungsfähigere Versionen der getesteten Modelle herausgegeben, die möglicherweise die identifizierten Beschränkungen der untersuchten Modelle nicht mehr aufweisen. Des Weiteren hängt die Qualität des generierten Codes und damit die Lösungen stark von der Formulierung der Prompts und der Expertise des Nutzers ab, in der Untersuchung, gab es keinen Vergleich von Ergebnissen über verschiedene Prompt-Anwender. Aus diesem Grund bleibt offen, ob und inwiefern die Ergebnisse von Anwender zu Anwender variieren. Da sich die Untersuchungen in der Arbeit auf Tests in einer kontrollierten Umgebung mit vorab gezielt definierten Szenarien beziehen bleibt unklar, inwiefern diese Ergebnisse repräsentativ für reale Situationen sind und ob die Leistungsfähigkeit der Modelle so in praxisnahen Umgebungen richtig eingeschätzt werden kann. Die angeführten Limitationen entsprechen aber typischen Restriktionen vergleichender Modellstudien und beeinträchtigen nicht die Validität der Kernaussagen.

5 Conclusio und Ausblick

Diese Masterarbeit hat die Fähigkeiten verschiedener Large Language Models zur Generierung von Python-Code für Geodatenverarbeitungsaufgaben in drei Testszenarien untersucht und evaluiert. Die Ergebnisse bestätigen die eingangs aufgestellte Forschungshypothese, wonach sich die getesteten LLMs signifikant hinsichtlich ihrer Fähigkeit unterscheiden, korrekten Python-Code für räumliche Analyseaufgaben zu generieren. Dabei zeigten sich Unterschiede sowohl in den ermittelten Accuracy-, als auch den Pass@k-Werten zwischen den Modellen, aber auch zwischen den Testszenarien mit variierender Komplexität bezüglich Mehrstufigkeit und Logikanforderung der Aufgaben. Die Analyse des generierten Codes verdeutlichte, dass die Modelle grundsätzlich in der Lage sind, funktionalen Code für spezifische georäumliche Aufgaben zu erstellen, jedoch stark in ihrer Effektivität variieren. Das Modell GPT o1-preview lieferte eine hohe Genauigkeit und konnte logische Fehler in einem Szenario gegenüber den anderen Modellen vermeiden, hierbei zeigte sich der spezielle Ansatz dieses Modells zur strukturierten Problemlösung als vorteilhaft. Das spezifisch auf Code trainierte Modell Deepseek Coder und Claude Sonnet 3.5 dagegen fielen in der Leistung gegenüber o1 ab und hatten Probleme mit der korrekten Implementierung von komplexen räumlichen Operationen. Ein weiteres zentrales Ergebnis war, das die häufigsten Fehler der Modelle die Verwendung von veralteten Methoden oder die fehlerhafte Behandlung von verschiedenen Typen von Geodaten waren, was darauf hindeutet, dass die Modelle zwar allgemeine Programmierprinzipien gut umsetzen können, aber Schwierigkeiten haben spezifische GIS-Bibliotheken korrekt anzuwenden. Die bedeutet, dass durch dynamische API-Änderungen die Modelle kontinuierliches Model-Retraining mit aktuellen Bibliotheken benötigen. Generell zeigen die Ergebnisse, dass LLM`s ein großes Potenzial für die Automatisierung der Geodatenverarbeitungsaufgaben bergen. Für eine Ausweitung meiner Forschung wäre eine Untersuchung von einem breiteren Feld von Modellen, welche verschiedene technische Ansätze verfolgen und auf unterschiedlichen Trainingsdaten basieren, denkbar, um aussagekräftigere Ergebnisse zu erhalten. Mit den gewonnenen Erkenntnissen aus diesen Studien könnte dann ein eigenes Modell auf Code trainiert werden, der Bibliotheken verwendet, speziell dafür optimiert, Geodaten zu verarbeiten. Dies wird auch im Forschungspapier: „Evaluation of Code LLMs on Geospatial Code Generation“ vorgeschlagen. Zudem wäre durch eine größere Bandbreite an verschiedenen Arten von Testszenarien eine höhere Generalisierbarkeit der Ergebnisse sichergestellt und Over Fitting lässt sich besser vermeiden. Zum Beispiel lässt sich die Variabilität der Datentypen erweitern, statt nur Vektordaten könnten auch Rasterdaten miteinbezogen werden. Des Weiteren würde eine größere Anzahl an Komplexitätsstufen der Testszenarien eine differenziertere Analyse der Fähigkeiten der Modelle erlauben. Die Einsatzfelder könnten von klassischen

Geodatenverarbeitungswerkzeugen wie Buffern, bis hin zu K.I-basierten Werkzeugen, wie die Objektklassifikation in Satellitenbildern, reichen. Durch verschiedene gestaffelte Schwierigkeitsniveaus, die von sehr einfachen Scripts bis hin zu mehrstufigen Workflows reichen, würde sich gezielt untersuchen lassen, wo die Modelle an ihre Grenzen stoßen. Für zukünftige Studien bietet es sich an zusätzlich Few-Shot-, statt nur Zero-Shot Techniken bei der Prompt-Übergabe an die Modelle zu verwenden, die möglicherweise die Ergebnisse verbessern können. Ein weiterer spannender Ansatz wäre, den Modellen die Möglichkeit zu geben, ihren eigenen generierten Code zu verbessern, indem per Prompt, die aufgetretenen Code-Fehlermeldung zur Verfügung gestellt wird oder, indem man Fehler-Feedbackloops integriert und so das Modell durch iterative Schleifen den eigenen Output schrittweise verbessert. Generell wird gerade an einer neuen Modellarchitektur geforscht, die nicht explizit eine Abfolge von logischen Schritten in natürlicher Sprache generiert, was zu weniger effizienten Berechnungen führt, weil viele Tokens nur für die sprachliche Kohärenz da sind, wie es bei der klassischen Transformer-Architektur der Fall ist. Sondern bei diesem neuen Ansatz werden stattdessen Reasoning-Schritte in einer abstrakten, kontinuierlichen Repräsentation durchgeführt, wodurch generell die planerischen Fähigkeiten des Modells verbessert werden sollen (Hao et al., 2024, S. 1–11) und sich damit auch stark auf die Fähigkeit zur Planung von Geodatenverarbeitungs-Workflows auswirken könnte. Letztlich wäre ein weiterer Ansatz zu testen, wie sich das Modell bei Datenlücken oder fehlerhaften Ausgangsdaten verhält, um noch näher an die praktische Anwendung zu kommen, da unbereinigte Ausgangsdaten in der Realität häufig vorkommen. So könnte erforscht werden, ob das Modell mit diesen erschwerten Bedingungen beispielsweise durch Plausibilitätsprüfungen oder automatische Anpassungen und Bereinigungen umgehen kann. Dadurch könnte der praktische Wert der Modelle und die Zuverlässigkeit in realen GIS-Anwendungen noch akkurater ermittelt werden.

6 Literaturverzeichnis

———. *Class: edit*. (2025, 26. Januar). Verfügbar unter: https://qgis.org/pyqgis/master/core/edit.html#module-edit

———. *Class: QgsDistanceArea*. (2025, 26. Januar). Verfügbar unter: https://qgis.org/pyqgis/master/core/QgsDistanceArea.html

———. *Class: QgsDistanceArea*. (2025, 26. Januar). Verfügbar unter: https://qgis.org/pyqgis/master/core/QgsDistanceArea.html

———. *Class: QgsGeometry*. (2025, 26. Januar). Verfügbar unter: https://qgis.org/pyqgis/3.40/core/QgsGeometry.html#qgis.core.QgsGeometry.asMultiPolyline

———. *Class: QgsMapLayer*. (2025, 26. Januar). Verfügbar unter: https://qgis.org/pyqgis/3.40/core/QgsMapLayer.html

———. *Class: QgsProcessingFeatureSourceDefinition*. (2025, 26. Januar). Verfügbar unter: https://qgis.org/pyqgis/3.40/core/QgsProcessingFeatureSourceDefinition.html

———. *Class: QgsProject*. (2025, 26. Januar). Verfügbar unter: https://qgis.org/pyqgis/3.40/core/QgsProject.html

———. *Class: QgsRectangle*. (2025, 26. Januar). Verfügbar unter: https://qgis.org/pyqgis/3.34/core/QgsRectangle.html

———. *Class: QgsSpatialIndex*. (2025, 26. Januar). Verfügbar unter: https://qgis.org/pyqgis/3.40/core/QgsSpatialIndex.html

———. *Class: QgsVectorLayer*. (2025, 26. Januar). Verfügbar unter: https://qgis.org/pyqgis/3.40/core/QgsVectorLayer.html

———. *Vector overlay — QGIS documentation*. (2025, 1. Februar). Verfügbar unter: https://docs.qgis.org/3.34/en/docs/user_manual/processing_algs/qgis/vectoroverlay.html#line-intersection

———. *QRect Class | Qt Core 6.8.1*. (2025, 8. Januarn). Verfügbar unter: https://doc.qt.io/qt-6/qrect.html

Amaratunga, T. (2023). *Understanding Large Language Models. Learning Their Underlying Concepts and Technologies* (1st ed. 2023). Berkeley, CA: Apress; Imprint Apress. https://doi.org/10.1007/979-8-8688-0017-7

Anthropic. (2024a). *Models - Anthropic*. Verfügbar unter: https://docs.anthropic.com/en/docs/about-claude/models

Anthropic. (2024, 5. Novemberb). *Models - Anthropic*. Verfügbar unter: https://docs.anthropic.com/en/docs/about-claude/models

Bai, Y., Kadavath, S., Kundu, S., Askell, A., Kernion, J., Jones, A. et al. (2022, 15. Dezember). *Constitutional AI: Harmlessness from AI Feedback*. Verfügbar unter: http://arxiv.org/pdf/2212.08073

Bartelme, N. (2005). *Geoinformatik. Modelle " Strukturen " Funktionen* (SpringerLink Bücher, 4., vollständig überarbeitete Auflage). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/b138747

The Big Benchmarks Collection - a open-llm-leaderboard Collection. (2025, 22. Januarb). Verfügbar unter: https://huggingface.co/collections/open-llm-leaderboard/the-big-benchmarks-collection-64faca6335a7fc7d4ffe974a

Brown, T. B., Mann, B. [Benjamin], Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P. et al. (2020, 28. Mai). *Language Models are Few-Shot Learners*. Verfügbar unter: http://arxiv.org/pdf/2005.14165v4

Chatbot Arena Leaderboard - a Hugging Face Space by lmarena-ai. (2024, 12. Novembera). Verfügbar unter: https://huggingface.co/spaces/lmarena-ai/chatbot-arena-leaderboard

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J. et al. (2021, 7. Juli). *Evaluating Large Language Models Trained on Code*. Verfügbar unter: http://arxiv.org/pdf/2107.03374v2

Chin-Yew Lin. *ROUGE: A Package for Automatic Evaluation of Summaries.* In Proceedings of the Workshop on Text Summarization Branches Out (WAS 2004). Verfügbar unter: https://aclanthology.org/W04-1013.pdf

The Claude 3 Model Family: Opus, Sonnet, Haiku. (2024b). Verfügbar unter: https://www.semanticscholar.org/paper/The-Claude-3-Model-Family%3A-Opus%2C-Sonnet%2C-Haiku/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627

Gao, S., Hu, Y. & Li, W. (2024). *Handbook of geospatial artificial intelligence* (First edition). Boca Raton, FL: CRC Press. https://doi.org/10.1201/9781003308423

GISGeography (2022, 2. Januar). Which Programming Language Should You Learn in GIS? *GIS Geography*. Verfügbar unter: https://gisgeography.com/gis-programming/

Gramacki, P., Martins, B. & Szymański, P. (2024, 6. Oktober). *Evaluation of Code LLMs on*

Geospatial Code Generation. Verfügbar unter: http://arxiv.org/pdf/2410.04617v1

Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W. et al. (2024, 25. Januar). *DeepSeek-Coder: When the Large Language Model Meets Programming -- The Rise of Code Intelligence*. Verfügbar unter: http://arxiv.org/pdf/2401.14196

Hao, S., Sukhbaatar, S., Su, D., Li, X., Hu, Z., Weston, J. et al. (2024, 9. Dezember). *Training Large Language Models to Reason in a Continuous Latent Space*. Verfügbar unter: http://arxiv.org/pdf/2412.06769

Innovative Data Energy Applications. (2025, 7. Februar). Verfügbar unter: https://maps.nrel.gov/

Janowicz, K., Gao, S., McKenzie, G., Hu, Y. & Bhaduri, B. (2020). GeoAI: spatially explicit artificial intelligence techniques for geographic knowledge discovery and beyond. *International Journal of Geographical Information Science*, *34*(4), 625–636. https://doi.org/10.1080/13658816.2019.1684500

Lange, N. de. (2020). *Geoinformatik in Theorie und Praxis : Grundlagen von Geoinformationssystemen, Fernerkundung und digitaler Bildverarbeitung* (4th ed. 2020). Berlin Heidelberg: Springer Berlin Heidelberg Imprint: Springer Spektrum.

Li, Z. [Zhenlong] & Ning, H. (2023). Autonomous GIS: the next-generation AI-powered GIS. *International Journal of Digital Earth*, *16*(2), 4668–4686. https://doi.org/10.1080/17538947.2023.2278895

Liu, J., Xia, C. S., Wang, Y. & Zhang, L. (2023, 2. Mai). *Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation*. Verfügbar unter: http://arxiv.org/pdf/2305.01210

Mai, G., Huang, W., Sun, J., Song, S., Mishra, D., Liu, N. et al. (2023, 13. April). *On the Opportunities and Challenges of Foundation Models for Geospatial Artificial Intelligence*. Verfügbar unter: http://arxiv.org/pdf/2304.06798v1

Manvi, R., Khanna, S., Mai, G., Burke, M., Lobell, D. & Ermon, S. (2023, 10. Oktober). *GeoLLM: Extracting Geospatial Knowledge from Large Language Models*. Verfügbar unter: http://arxiv.org/pdf/2310.06213v2

Mooney, P., Cui, W., Guan, B. & Juhász, L. Towards Understanding the Geospatial Skills of ChatGPT. In *Newsam, Yang et al. (Hg.) 2023 – Proceedings of the 6th ACM* (S. 85–94). https://doi.org/10.1145/3615886.3627745

Papineni, K., Roukos, S., Ward, T. & Zhu, W.-J. (2001). BLEU. In P. Isabelle (Hrsg.), *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02* (S. 311). Morristown, NJ, USA: Association for Computational Linguistics.

Peter F. Fisher. (1989). Expert System Applications in Geography. *AREA Royal Geographical Society with IBG*, (Vol. 21, No. 3), 279–287.

QGIS Österreich. (2021, 13. Oktober). Verfügbar unter: https://qgis.at/

Sabbatino, M. 2018. *Global Oil & Gas Features Database*. https://doi.org/10.18141/1427300

Tao, R. & Xu, J. (2023). Mapping with ChatGPT. *ISPRS International Journal of Geo-Information*, *12*(7), 284. https://doi.org/10.3390/ijgi12070284

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N. et al. (2017, 12. Juni). *Attention Is All You Need*. Verfügbar unter: http://arxiv.org/pdf/1706.03762v7

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F. et al. (2022, 28. Januar). *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. Verfügbar unter: http://arxiv.org/pdf/2201.11903

What are the Top Programming Languages in the GIS World? – GoGeomatics. (2025, 17. Januar). Verfügbar unter: https://gogeomatics.ca/what-are-the-top-programming-languages-in-the-gis-world/?utm_source=chatgpt.com

Xie, Y., Wang, Z., Mai, G., Li, Y., Jia, X., Gao, S. et al. (2023). Geo-Foundation Models: Reality, Gaps and Opportunities. In M. L. Damiani, M. Renz, A. Eldawy, P. Kröger & M. A. Nascimento (Hrsg.), *Proceedings of the 31st ACM International Conference on Advances in Geographic Information Systems* (S. 1–4). New York, NY, USA: ACM.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y. et al. (2023, 18. Mai). *Tree of Thoughts: Deliberate Problem Solving with Large Language Models*. Verfügbar unter: http://arxiv.org/pdf/2305.10601

7 Anhang

7.1 Beispiel Testprotokoll

Testprotokoll Testszenario 2

Model: GPT o1 Preview/ Prompt: 5

Eingabe-Prompt:

Erstelle einen Python Code für die QGIS Python Konsole:

Im aktuellen QGIS-Projekt gibt es einen Layer „gebaeude_kloburg“ mit Polygon-Features, die Gebäude darstellen.

1. Erstelle einen neuen Layer „Versorgungsnetzwerk“ im Koordinatensystem: EPSG: 4326, mit Line-Features, der Stromleitungen zwischen den Gebäuden repräsentiert.

- Diese Leitungen sollen die nächstgelegenen Gebäude miteinander verbinden und dürfen nicht länger als 500 Meter sein.

2. Wenn ein Gebäude mehr als 3 Verbindungen hat, erstelle einen neuen Layer „Umspannstationen“ im Koordinatensystem: EPSG: 4326, der an diesem Gebäude ein Symbol (Punkt-Feature) darstellt.

Generierter Code:

```
# import der notwendigen Module
from qgis.core import (
    QgsProject,
    QgsFeature,
    QgsGeometry,
    QgsPoint,
    QgsPointXY,
    QgsSpatialIndex,
    QgsCoordinateReferenceSystem,
```

```
    QgsCoordinateTransform,
    QgsField,
    QgsFields,
    QgsVectorLayer,
    QgsWkbTypes,
    QgsDistanceArea
)
from PyQt5.QtCore import QVariant

# 1. Im QGIS geladener Layer "gebaeude_kloburg" abrufen und in Variable speichern
gebaeude_layer = QgsProject.instance().mapLayersByName('gebaeude_kloburg')[0]

# 2. Quell- und Ziel-Koordinatensystem (EPSG:4326) in Variable speichern
source_crs = gebaeude_layer.crs()
target_crs = QgsCoordinateReferenceSystem('EPSG:4326')

# 3. Erstellung Koordinatentransformator in Variable
coord_transform = QgsCoordinateTransform(source_crs, target_crs, QgsProject.instance())

# 4. Vorbereitung der Sammlung von Zentroid-Features und räumlicher Index
centroid_features = []
spatial_index = QgsSpatialIndex()
feature_centroids = {}  # Schlüssel: Feature-ID, Wert: Zentroid-Punkt
centroid_feature_dict = {}  # Schlüssel: fid, Wert: Zentroid-Feature

# 5. Erstellung Zentroiden, transformmation zu EPSG:4326, Aufbau räumlicher Index
for feature in gebaeude_layer.getFeatures():
    geom = feature.geometry()
    centroid = geom.centroid()
    centroid.transform(coord_transform)
```

```
    centroid_feature = QgsFeature()
    centroid_feature.setGeometry(centroid)
    centroid_feature.setAttributes([feature.id()])
    centroid_feature.setId(feature.id())  # Setzen Sie die ID auf die ursprüngliche Feature-ID
    centroid_features.append(centroid_feature)
    centroid_feature_dict[feature.id()] = centroid_feature
    spatial_index.addFeature(centroid_feature)
    feature_centroids[feature.id()] = centroid_feature.geometry().asPoint()

# 6. Vorbereitung des Distanzberechners
distance_calculator = QgsDistanceArea()
distance_calculator.setSourceCrs(target_crs, QgsProject.instance().transformContext())
distance_calculator.setEllipsoid('WGS84')

# 7. Finden von Verbindungen zwischen Gebäuden innerhalb von 500 Metern
connections = {}  # Schlüssel: Tupel von sortierten fids, Wert: Distanz
connection_counts = {}  # Schlüssel: fid, Wert: Anzahl der Verbindungen

for feature in centroid_features:
    fid = feature.id()
    point = feature.geometry().asPoint()
    # Nahegelegene Features mithilfe des räumlichen Index abrufen
    nearby_ids = spatial_index.nearestNeighbor(point, 10)  # Bis zu 10 nächste
    for near_fid in nearby_ids:
        if near_fid == fid:
            continue  # Überspringe sich selbst
        # Nahegelegenes Zentroid-Feature abrufen
        near_feature = centroid_feature_dict[near_fid]
        near_point = near_feature.geometry().asPoint()
        # Distanz berechnen
```

```
    dist = distance_calculator.measureLine(point, near_point)
    if dist <= 500:

        connection = tuple(sorted([fid, near_fid]))
        if connection not in connections:
            connections[connection] = dist
            # Aktualisieren der Verbindungsanzahl
            connection_counts[fid] = connection_counts.get(fid, 0) + 1
            connection_counts[near_fid] = connection_counts.get(near_fid, 0) + 1
    else:
        continue

# 8. Erstellung von Linien-Layer "Versorgungsnetzwerk" in EPSG:4326
line_layer = QgsVectorLayer("LineString?crs=EPSG:4326", "Versorgungsnetzwerk", "memory")
line_provider = line_layer.dataProvider()

# Felder definieren
fields = QgsFields()
fields.append(QgsField("fid1", QVariant.Int))
fields.append(QgsField("fid2", QVariant.Int))
fields.append(QgsField("distance_m", QVariant.Double))
line_provider.addAttributes(fields)
line_layer.updateFields()

# Linien-Features erstellen
line_features = []

for connection, distance in connections.items():
    fid1, fid2 = connection
    point1 = feature_centroids[fid1]
```

```
    point2 = feature_centroids[fid2]
    line = QgsGeometry.fromPolylineXY([QgsPointXY(point1), QgsPointXY(point2)])
    line_feature = QgsFeature()
    line_feature.setGeometry(line)
    # Attribute setzen
    line_feature.setAttributes([fid1, fid2, distance])
    line_features.append(line_feature)

# Features zum Linien-Layer hinzufügen
line_provider.addFeatures(line_features)
line_layer.updateExtents()

# Linien-Layer zum Projekt hinzufügen
QgsProject.instance().addMapLayer(line_layer)

# 9. Identifizieren der Gebäude mit mehr als 3 Verbindungen
umspann_fids = [fid for fid, count in connection_counts.items() if count > 3]

# 10. Erstellung von Punkt-Layer "Umspannstationen" in EPSG:4326
umspann_layer = QgsVectorLayer("Point?crs=EPSG:4326", "Umspannstationen", "memory")
umspann_provider = umspann_layer.dataProvider()

# Felder definieren
fields = QgsFields()
fields.append(QgsField("fid", QVariant.Int))
umspann_provider.addAttributes(fields)
umspann_layer.updateFields()

# Punkt-Features erstellen
umspann_features = []
```

```
for fid in umspann_fids:
    point = feature_centroids[fid]
    point_feature = QgsFeature()
    point_feature.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(point)))
    point_feature.setAttributes([fid])
    umspann_features.append(point_feature)

# Features zum Umspann-Layer hinzufügen
umspann_provider.addFeatures(umspann_features)
umspann_layer.updateExtents()

# Umspann-Layer zum Projekt hinzufügen
QgsProject.instance().addMapLayer(umspann_layer)
```

Ausführung:

Erwartetes Ergebnis:

Erstellung Layer: „Versorgungsnetzwerk“ mit Line-Features kürzer als 500 Meter

Erstellung Layer: „Umspannstationen“ mit Punkt-Features

Tatsächliches Ergebnis:

Wie erwartet

Unit Tests Ergebnis:

Layer 'Versorgungsnetzwerk' wurde gefunden.

Koordinatensystem des 'Versorgungsnetzwerk'-Layers ist korrekt (EPSG:4326).

Der 'Versorgungsnetzwerk'-Layer enthält insgesamt 2156 Features.

2156 von 2156 Verbindungen im 'Versorgungsnetzwerk'-Layer sind kürzer oder gleich 500 Meter.

0 von 2156 Verbindungen sind länger als 500 Meter.

Layer 'Umspannstationen' wurde gefunden.

Koordinatensystem des 'Umspannstationen'-Layers ist korrekt (EPSG:4326).

Der 'Umspannstationen'-Layer enthält insgesamt 393 Features.

393 von 393 Gebäuden haben Umspannstationen (mehr als 3 Verbindungen).

Bewertung:

Syntaktische Korrektheit:

ja

Ausführbarkeit:

ja

Funktionale Anforderungen erfüllt:

ja

7.2 Berechnungsschritte der Pass@k-Metrik

7.2.1 GPT o1 preview Berechnungsschritte

Formel:

$$\text{pass@k} = 1 - \prod_{i=n-c+1}^{n} \left(1 - \frac{k}{i}\right)$$

Task 1: (n=20, c=4, k=5)

Parameter:

- n = 20: Gesamtzahl der generierten Lösungen.

- c = 4: Anzahl der korrekten Lösungen.

- k = 5: Anzahl der getesteten Lösungen.

Berechnung:

$$\prod_{i=20-4+1}^{20}\left(1-\frac{5}{i}\right)=\left(1-\frac{5}{17}\right)\cdot\left(1-\frac{5}{18}\right)\cdot\left(1-\frac{5}{19}\right)\cdot\left(1-\frac{5}{20}\right)$$

$$=0.7059\cdot 0.7222\cdot 0.7368\cdot 0.75\approx 0.2817$$

Berechnung von pass@k:

pass@k=1−0.2817≈0.7183

Task 2: (n=20, c=2, k=5)

Parameter:

- n = 20, c = 2, k = 5.

Berechnung:

$$\prod_{i=20-2+1}^{20}\left(1-\frac{5}{i}\right)=\left(1-\frac{5}{19}\right)\cdot\left(1-\frac{5}{18}\right)$$

$$=0.7368\cdot 0.7222\approx 0.5315$$

Berechnung von pass@k:

pass@k = 1 - 0.5315 ≈ 0.4474

Task 3: (n=20, c=4, k=5)

Parameter:

- n = 20, c = 4, k = 5.

Berechnung:

$$\prod_{i=20-4+1}^{20}\left(1-\frac{5}{i}\right)=\left(1-\frac{5}{17}\right)\cdot\left(1-\frac{5}{16}\right)$$

$$=0.7059\cdot 0.6875\approx 0.4944$$

Berechnung von pass@k:

pass@k = 1 - 0.4944 ≈ 0.7183

Durchschnittliche Pass@k-Metrik:

Berechnung des Durchschnitts:

Durchschnitt = (0.25 + 0.4474 + 0.7183) / 3 ≈ 0.4719

7.2.2 Deepseek Coder Berechnungsschritte

Formel

$$\text{pass@k}=1-\prod_{i=n-c+1}^{n}\left(1-\frac{k}{i}\right)$$

Task 1: (n=20, c=1, k=5)

Parameter:

- n = 20: Gesamtzahl der generierten Lösungen.
- c = 1: Anzahl der korrekten Lösungen.
- k = 5: Anzahl der getesteten Lösungen.

Berechnung der Produktkomponente:

$$\prod_{i=20-1+1}^{20} \left(1 - \frac{5}{i}\right) = \left(1 - \frac{5}{20}\right) \cdot \left(1 - \frac{5}{19}\right)$$

$$= 0.75 \cdot 0.7368 \approx 0.5526$$

Task 2: (n=20, c=0, k=5)

n=20, c= 0, c = 0, k = 5

Berechnung der Produktkomponente:

$$\prod_{i=20-0+1}^{20} \left(1 - \frac{5}{i}\right) = \left(1 - \frac{5}{21}\right) \cdot \left(1 - \frac{5}{20}\right) \cdot \ldots$$

Berechnung von Pass@k:

Keine korrekten Lösungen vorhanden, daher:

pass@k=0

Task 3: (n=20, c=3, k=5)

Parameter:

N = 20, c = 3, k = 5.

Berechnung der Produktkomponente:

$$\prod_{i=20-3+1}^{20}\left(1-\frac{5}{i}\right)=\left(1-\frac{5}{18}\right)\cdot\left(1-\frac{5}{19}\right)$$

$$=0.7222\cdot 0.7368\approx 0.5323$$

Berechnung von pass@k:

pass@k=1 −0.5323 ≈ 0.4677

Durchschnitliche Pass@k-Metrik:

$$\text{Durchschnitt}=\frac{\text{Summe aller Pass@k-Werte}}{\text{Anzahl der Tasks}}$$

$$\text{Durchschnitt}=\frac{0.25+0+0.4677}{3}\approx 0.2392$$

7.2.3 Claude Sonnet 3.5 Berechnungsschritte

Formel:

$$\text{pass@k}=1-\prod_{i=n-c+1}^{n}\left(1-\frac{k}{i}\right)$$

Task 1: (n=20, c=0, k=5)

Parameter:

n=20: Gesamtzahl der generierten Lösungen.

c=0: Anzahl der korrekten Lösungen.

k=5: Anzahl der getesteten Lösungen.

Berechnung der Produktkomponente:

Keine korrekten Lösungen, daher (c = 0) und es ergibt sich:

$$\prod_{i=20-0+1}^{20} \left(1 - \frac{5}{i}\right) = 1$$

Berechnung von pass@k:

pass@k = 1 – 1 = 0.0

Task 2: (n = 20, c = 0, k = 5)

ident mit Task 1

pass@k = 0.0

Task 3: (n=20, c=3, k=5)

n=20: Gesamtzahl der generierten Lösungen.

c=3: Anzahl der korrekten Lösungen.

k=5: Anzahl der getesteten Lösungen.

Berechnung der Produktkomponente:

$$\prod_{i=20-3+1}^{20} \left(1 - \frac{5}{i}\right) = \left(1 - \frac{5}{18}\right) \cdot \left(1 - \frac{5}{19}\right) \cdot \left(1 - \frac{5}{20}\right)$$

$$= 0.7222 \cdot 0.7368 \cdot 0.75 \approx 0.3991$$

Berechnung von pass@k:

pass@k = 1 – 0.3391 ≈ 0.6009

$$\text{Durchschnitt} = \frac{\text{Summe aller Pass@k-Werte}}{\text{Anzahl der Tasks}}$$

$$\text{Durchschnitt} = \frac{0.0 + 0.0 + 0.6009}{3} \approx 0.2003$$

8 Erklärung zur Nutzung von generativer KI

Für das Verfassen dieser Masterarbeit wurde ChatGPT und weitere LLM‘s eingesetzt, um die Formulierungen stilistisch, grammatikalisch und orthografisch zu optimieren und um das Grundstruktur von Tabellen zu erstellen.